An Implementation of Domains and Keys in SQL

A Thesis

presented in fulfilment of

the requirements for the degree

Master of Philosophy

at

Massey University

John Lindsay Alexander

1987

ABSTRACT

The relational Data Base Management System model has steadily acquired increasing acceptance over the years since it was first introduced in 1970, yet few - if any - of the many relational products currently available support the fundamental concepts of domains and the integrity constraints of primary and foreign keys. Over more recent years the SQL Relational Database Sub-Language has found most favour among users and vendors alike, and a standard for SQL has recently been produced. This standard provides no support for domains or foreign keys, and only indirect support for primary keys.

This thesis first reviews MURDER, the relational dabatase management system used for teaching purposes at Massey, and then describes an implementation of the domain and key concepts, highlighting some of the problem areas still to be resolved. Also described is an implementation of the query and update facilities of SQL, including some extensions which it is claimed increase its functionality. Finally, refinements to the language definition are suggested, to remove some redundancies and ambiguities.

Acknowledgements

I wish to express my gratitude to

my supervisors June Verner and Chris Phillips, for their encouragement, advice and support;

Elizabeth Kemp, for her patient forbearance when the system kept changing, and for many helpful suggestions;

the many students, known and unknown, who never failed to activate the most obscure and insidious bugs; and

my wife Shirley, without whose acceptance of the unsocial habits such an undertaking inevitably develops, nothing would have been possible.

CONTENTS

ILLUSTRATIONS

ALGORITHMS

CHAPTER 1

INTRODUCTION

1.1. Overview

1.1.1. Background

Engles[72] defined a database as

> "... the collection of stored operational data used by the application systems of some particular enterprise."

Databases have been around since the clay-tablet days of early civilization, and have progressed - and expanded - through pen and paper to the age of the computer. In the late sixties, with the development of electronic data processing, the idea of having a system program to manage stored data emerged, so that all aspects of an enterprise might be served from one common resource. This is the Data Base Management System, or DBMS.

Since then three main approaches to the architecture of a DBMS have been developed more or less concurrently. These were

hierarchical e.g. IBM's Information Management System, which has been historically popular, and was based on existing storage strategies using trees;

network — based on the April 1971 report of CODASYL's Data Base Task Group (DBTG[71]), which also used existing storage strategies; and

relational — introduced in Codd[70], and based on the mathematical theory of relations.

Whereas the first two of these approaches were largely empirical, only the third had a sound theoretical basis. Further, these first two approaches require the major relationships between data sets (which correspond approximately to files in traditional systems) to be built into the data model. Those accesses which are anticipated and allowed for by the database designer may be executed quickly and efficiently; any which are not anticipated, or which are not included either because they would compromise other accesses or are only required relatively infrequently, may need detailed knowledge of the database design, often involve complex programming, and may be expensive in terms of resources.

In contrast, for a relational system all data interrelationships are implicit, so that all expressions may be formed in basically the same way without the user needing to have detailed knowledge of the database design.

1.1.2. Database Teaching at Massey

In the mid 70s a small relational DBMS, called simply RDBMS, was implemented at Massey as an honours project. At that time there was little available which could give

students experience of a relational environment, and it was felt to be unsatisfactory to teach such a practical subject without practical experience. It was used for teaching here until 1984, and is still in use at a number of sites both in New Zealand and Australia. Initially RDBMS provided a reasonably complete relational algebra, with the divide operation slightly restricted. This was then gradually extended to include a number of facilities for relational calculus. (Relational algebra and calculus are the two main approaches to formulating query expressions in a relational system. They are further explained in Chapters 4 and 5 respectively.)

However, when it was attempted to extend the range of calculus expressions which RDBMS could handle, it was found that the approach originally taken made this impractical. To overcome this a new system was proposed for Massey's Prime 750.

1.1.3. The MURDER System

Initially the system included simple utilities to open and close a database, list relations etc., together with an algebraic interface. A calculus interface was added later. Extensive modifications to the initial approach were needed, to improve execution speed and to reduce resource requirements. (The modifications required to optimise the calculus are described in Chapter 5.)

In particular, the decision to implement a memory-resident database had to be altered. (A resident approach had been

chosen to simplify the task by eliminating the problems of disk access and management, but it was found that in unoptimised calculus queries the program rapidly used up the available memory.) The resulting system, called MURDER (for Massey University Relational Database Environment), has been in use for teaching since 1984 at Massey, and has been introduced to two sites in Australia.

With the optimised approach to evaluating calculus queries it again became possible to consider a memory-resident system.

When the Computer Science Department acquired its Vax 11/750s, the system was translated into C for this machine (running under ULTRIX, DEC's version of UNIX). This version of MURDER is memory resident, and has been used successfully for postgraduate teaching, as well as the development of new facilities, including those which are described in this document. Many of these extensions have since been translated back into PL/1G for the undergraduate version running on the Prime.

1.2. Project Outline

The relational approach has clearly found most favour, and of the variety of interfaces which have appeared the most widely used is SQL. As a standard SQL-like interface would soon appear, it was decided to implement a SQL query interface to the MURDER system, and at the same time take the opportunity to investigate two of the major omissions from current implementations, domains and keys.

Increasingly within the literature there is an insistence that these two are both required by theory and necessary for the integrity of the database, but they commonly do not appear in released products. Codd[85] argues that in the 80s a system must support these concepts (as well as a number of others) if it is to be classified as relational. The significance of domains and keys is outlined in the next chapter.

Thus there are three main parts to the work undertaken:

- investigation and implementation of domains;
- investigation and implementation of primary and foreign keys; and
- the provision of a SQL query interface. (SQL updates are also supported, largely because updates form a significant part of the previous point.)

The remainder of this introductory chapter specifies the notation used for MURDER syntax fragments.

Chapter 2 specifies the simple database used in examples in this document, and outlines the theory of the relational approach. Chapter 3 describes the basic implementation of MURDER, including the extensions to handle domains and keys. Much of Chapter 4, covering relational algebra, is a necessary background to the relational calculus of Chapter 5, and this in turn sets the scene for the SQL implementation. Chapter 6 outlines the history and development of SQL, and in Chapter 7 the SQL

implementation is described. Finally Chapter 8 presents a summary and some conclusions.

Appendix A outlines the C-like notation used to present most algorithms, and the remaining appendices give the syntax for SEQUEL, SEQUEL2 (the two versions which preceded SQL) and the MURDER interface.

1.3. Syntax

MURDER syntax throughout this document is in the form of b.n.f. described below:

::=	may be read 'is defined as'
UPCASE	denotes a required keyword
locase	denotes a token substituted by the user, or defined in a further production
[ ]	enclose optional parts of productions or tokens
|	separates alternative productions

Whenever one of the metacharacters [,] or | is required in a production, it is enclosed in single quotes; thus

```
logic_op ::= & |
             '|'
```

allows a 'logic_op' to be replaced by either the ampersand '&' or the vertical bar '|'; and

```
A [ B ]      represents either A or AB, whereas
A '[' B ']'  represents A[B]
```

CHAPTER 2

RELATIONAL DATABASE MANAGEMENT SYSTEMS

2.1. Example Database

Throughout this document examples are given using the SPJ database illustrated in Figure 2.1. The relations (primary key underlined) are

SUPPLIER(SNUM, SNAME, CITY)

PART(PNUM, PNAME, WEIGHT)

JOB(JNUM, JNAME, CITY)

and one or more of the following, depending on the relevance to the point being illustrated:

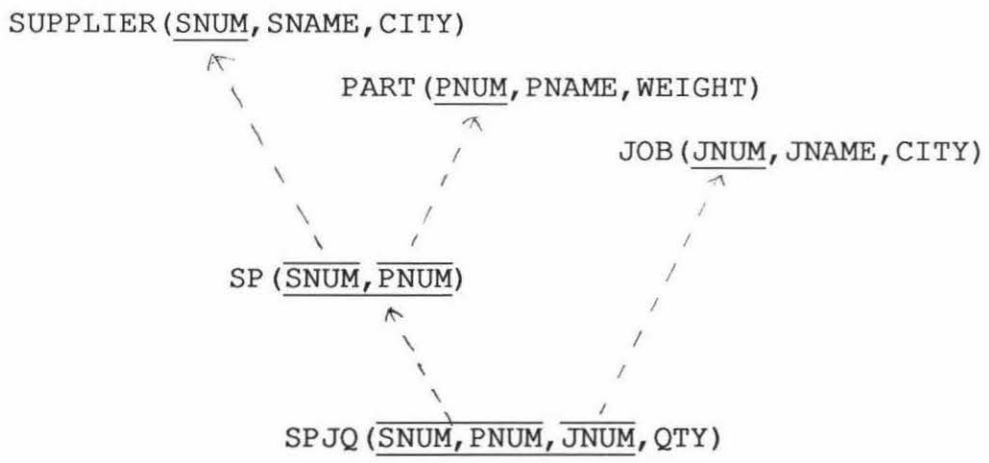


Figure 2.1 - SPJ Database

(Dotted lines indicate foreign key relationships)

SP(SNUM, PNUM)

PJ(PNUM, JNUM)

SPJQ(SNUM, PNUM, JNUM, QTY)

(In some of the examples, where the QTY attribute is irrelevant, the relation

SPJ(SNUM, PNUM, JNUM)

is used instead of SPJQ.)

As a convenient guide to the reader, attributes and domains have the same name, so that when two attributes have the same name they share a common domain. The domains and associated data types are

SNUM, PNUM, JNUM	: char(4)
SNAME, PNAME, JNAME	: char(20)
CITY	: char(8)
WEIGHT, QTY	: integer

2.2. Informal Introduction

The only visible data structure in a relational DBMS is the relation, which is rather like a flat file, with repeating groups eliminated in the process of normalization. This means that only 'atomic' fields are present, with each 'record' in the 'file' having a value for each 'field'. Records and fields are usually called tuples and attributes respectively. One could imagine the SUPPLIER relation from the example database defined in Section 2.1 as shown in Figure 2.2.

SUPPLIER

SNUM	SNAME	CITY
S1	ADAM	LONDON
S2	BRUCE	ATHENS
S3	CLARK	LONDON
..	..	..

Figure 2.2 - Realization of SUPPLIER Relation

It is important to remember that this representation need not reflect the internal organization; in particular neither the left-to-right ordering of the attributes, nor the top-to-bottom ordering of the tuples, is significant.

Each attribute has an associated 'domain', and its atomic values are drawn from the domain for that attribute. For each attribute the domain defines the set of all possible atomic values. In the example database of Section 2.1, both supplier and job numbers may be 4-character strings, for example. They could use the same domain, but making them separate domains will inhibit some improper use of data. Not all the elements of the domain need be used, nor indeed may all be realizable because of semantic restrictions on the data.

In place of the explicit relationships of the hierarchical and network approaches, the relational architecture substitutes implicit relationships formed by having attri-

butes from different relations share a common domain. In the example database both SUPPLIER and SP relations have an attribute SNUM from the same domain of supplier numbers. It is intended that each SP-tuple is implicitly associated with the SUPPLIER-tuple having the same value for the SNUM attribute. Such a SUPPLIER-tuple should exist - see the discussion of referential integrity in Section 2.4.2.

Here SNUM is the 'primary key' of the SUPPLIER relation, i.e. to identify a SUPPLIER-tuple uniquely it is sufficient to know the SNUM-value. Within the SP relation the SNUM attribute is a 'foreign key', i.e. the primary key of a different relation, as well as being part of the primary key. (Similarly PNUM is a foreign key of the PART relation.) Meaning is given to the data by associating tuples from two or more relations based on such foreign-to-primary key matchings.

2.3. Formal Definition

The more formal definition of a relational DBMS given here is based on material from Date[83a].

A relational database may be defined as the ordered pair (D, R), where

D ::= {{domain}}

R ::= {named_relation}

in which the double-braces '{{' and '}}' denote a non-empty set; single braces denote a set in the usual sense,

viz. possibly empty, without replication of elements, and with no ordering of the elements.

```
domain ::= domain_name {{domain_value}} order_flag
```

in which the static set of domain values is predefined; order flag (either true or false) indicates whether or not comparison operators, such as '<', are meaningful - it is always possible to impose an ordering, e.g. lexicographical order, on a set of values, but it may not be possible to say (for example) that one part number is greater than another in any inherently useful way.

A relation may be named or unnamed, and a named relation may be either real or virtual. An example of an unnamed relation is the result of a relational expression; a virtual relation has no permanent existence, but is derived at run-time from one or more real (or base) relations by evaluating a relational expression. A common term for a virtual relation is a 'view'. Thus the specifications are

```
named_relation ::= real_relation |
                   virtual_relation

real_relation ::= relation_name {{attribute}}
                    {{candidate_key}} {tuple}

virtual_relation ::= relation_name unnamed_relation

unnamed_relation ::= relational_expression

attribute ::= attribute_name domain_name
```

A relational expression is constructed according to the rules for available operators, such as the algebraic operators described in Chapter 4; every attribute in the

resulting relation may be unambiguously traced back to its origin in a real relation, and so has a name - possibly qualified by the relation name of this ultimate parent, although as Section 2.4 shows this convention may be inadequate for unique identification - and a domain.

An attribute consists of a name and an associated domain.

A candidate key is defined as 'a minimal set of attributes which uniquely identify tuples in the relation'. Then

primary_key ∈ {candidate_key}

{alternate_key} = {candidate_key} - {primary_key}

Since the tuples of a relation form a set, no two are the same, and hence the existence of at least one candidate key can be inferred. The set of all attributes is a key, although it may not be a candidate key because of the inclusion of 'minimal' in the definition. Any candidate key may be arbitrarily designated as the primary key, and any other candidate keys are then referred to as alternate keys.

Note that the result of a relational expression also has a non-empty set of attributes, at least one key - the concatenation of all attributes - and a (possibly empty) set of tuples.

Suppose that some relation R of degree or arity n has attributes

{(Ai, Di)}, i = 1, 2 ... n, n > 0;

then any tuple from the relation must be of the form

$$\{(Vi, Ai): Vi \in Di\}, i = 1, 2 \ldots n.$$

As the attributes form a set, and a set is unordered, it is strictly necessary for a tuple to consist of (value, attribute) pairs as shown here. If the convention that at any time the value of a relation fixes the order of its attributes is adopted, so that while it may not be significant it is at least consistent in both the relation definition and all tuples, then for practical purposes a tuple may be seen as an ordered collection of atomic values, with the appropriate domain for each value being implicitly specified by the ordinal position of the value in the tuple. With this assumption a tuple has the form

$$(V1, V2 \ldots Vn).$$

This simplifying assumption is made throughout this document.

There are some further complications concerning keys. No attribute in the primary key may accept the null value, assuming the DBMS allows nulls. (Null values are further discussed in Section 2.4.4; for the moment it will be sufficient to regard a null value as a recognisable value which indicates that no actual value for the attribute is currently stored.) This is implicit in the requirement that each tuple be uniquely identifiable by the value of (the attributes in) its primary key.

Relationships between real relations are specified through the concept of foreign keys. For the case when the primary key is a single attribute, if real relation R has primary key Ri defined on domain Di, and real relation S has an attribute Sj defined on the same domain Di, then for every S-tuple the value of attribute Sj must be either null or equal to the value of attribute Ri in some R-tuple. Attribute Sj is said to be a foreign key.

As an example, consider relations SUPPLIER and SP in the SPJ database. The primary key of the SUPPLIER relation is attribute SNUM, and within relation SP the SNUM attribute is a foreign key referencing the SUPPLIER relation. The definition above requires that either

- the value of the SNUM attribute in an SP-tuple be null (corresponding to the case where it is known that a particular part is required, but it has not yet been decided from which supplier the part is to be obtained); or

- if non-null, there must be a SUPPLIER-tuple with the same value for its SNUM attribute.

(This example is purely for the purposes of illustration, and is of course impractical - SNUM is part of the primary key of relation SP, and no attribute of a primary key may be null.)

The composite-key case is an extension of this, with Ri, Di and Sj being replaced by sets of attributes and

domains, and the requirement that the value of 'attribute' {Sj} be wholly null, or equal to the value of {Ri} from some R-tuple. Two sets of values are equal if and only if each corresponding pair of atomic values are equal.

2.4. Some Practical Problems

In this section some of the practical problems which must be solved before the theory outlined in Section 2.3 can be fully realized are outlined. However there is no attempt to give full solutions to these problems; the intention is rather to provide a broad background for the discussion in later chapters.

2.4.1. Domains

The theory outlined in the previous section requires that a new real relation may only be created explicitly, e.g. by something like

```
CREATE RELATION relation_name
                (attribute_name_1 : domain_name_1,
                 attribute_name_2 : domain_name_2,
                   ...
                 attribute_name_n : domain_name_n)
```

in which the domains named are already known to the DBMS. When this is done the validity of operators can be checked, e.g. for a comparison other than (in)equality both operands must come from the same - or at least a compatible - ordered domain; and similar checking can be done in the case of relational assignment.

Commonly, however, 'domain name' in the form suggested above is replaced by the weaker notion of 'data type', e.g. integer or char(4). Now comparisons are considered valid if the data types are the same, but this opens the way for nonsense queries: if WEIGHT and PRICE are both of data type real, then they can be compared or added (neither of which make sense), or multiplied (which might).

It should be pointed out that even with domains there are difficulties. If LENGTH and AREA are two domains with real data type, and LEN1 and LEN2 are two LENGTH values,

LEN1 + LEN2 is a LENGTH;
LEN1 * LEN2 is an AREA; and
2 * LEN2 could be either.

To resolve the first two cases requires that for each valid combination of a unary or binary operator and associated operand(s) the domain of the result be explicitly declared and stored as part of the database definition.

Arithmetic functions require similar definition. A count of any domain will normally not belong to the same domain, while the sum, maximum and minimum values will; and average may present its own problems if the domain is based on integers, since the average of a set of integers is often not an integer, and so is not in the same domain!

Some means must be found for resolving expressions using literals, probably by specifying the domain of the result. Mcleod[76] discusses this problem in some detail.

2.4.2. Keys

Two terms which are used to describe database integrity are 'entity integrity' and 'referential integrity'.

The former requires that it must be possible to specify each tuple in a relation uniquely, and as the primary key is the only tuple-level addressing mechanism in a relational database, it follows that each real relation must have a known primary key, and that this key may not accept null values for any of its attributes.

By referential integrity is meant the requirement (illustrated in the last example) that whenever a foreign key in a tuple of the referencing relation is not null, there exist in the referenced relation a tuple with primary key equal to that foreign key, i.e. referential integrity requires that foreign key relationships be known as part of the database definition. (Note that this means explicitly declaring at least some of the possible meaningful data inter-relationships, although it does not necessarily compromise the earlier claim that all expressions may be written in basically the same way.)

To meet these requirements, the syntax for creating a real relation must be extended to allow for the specification of the primary key and any foreign keys, e.g.

```
CREATE RELATION relation_name
                (attribute_domain_list)
      PRIMARY KEY attribute_name_list
  [ ; FOREIGN KEY attribute_name_list
                  REFERENCES real_relation ] ...
```

where the ellipsis '...' means one or more iterations. This is not ideal, as provision must be made for specifying additional foreign keys consequent upon subsequent declaration of new real relations; but it meets the immediate needs.

With the example database of Section 2.1, the SP relation would require definitions

```
CREATE DOMAIN SNUM CHAR(4)
CREATE DOMAIN PNUM CHAR(4)
 ...
CREATE RELATION SP
                (SNUM : SNUM,
                 PNUM : PNUM)
       PRIMARY KEY SNUM, PNUM;
       FOREIGN KEY SNUM REFERENCES SUPPLIER;
       FOREIGN KEY PNUM REFERENCES PART
```

The system can now check the validity of any tuples being added to relation SP, either individually or during the tuple set replacement of an assignment. But what about the deletion of, say, a supplier? Since the primary key attribute SNUM is a foreign key within relation SP, the referential integrity of the database would be lost if any SP-tuples include the same SNUM-value as the deleted supplier. Such tuples will either have to have the value of this attribute set to null (if nulls are permitted in the database and for this attribute) or be deleted, or else the original deletion must be rejected. In the present example setting the attribute value to null is not permissible because no attribute of a primary key may be null. Either the problem must be resolved by definition. e.g.

> When a foreign key is part of the primary key the tuple will be deleted; otherwise the attribute value will be set to null.

or the suggested syntax above needs further extension to resolve the system's dilemma when it is faced with a choice.

It may be that the delete operation should be rejected on semantic grounds because of problems arising in related relations; if the foreign key is in turn part of a primary key (as it is in the example) the whole operation must be considered at the next level; and so on.

The actions described here will also need to be carried out in the case of modification of a primary key, but the choice of alternative courses of action may differ for the two operations. Date[86] suggests that there are in fact three options:

restrict: the delete/update operation is restricted to the case where no foreign key references the target tuple, and is rejected otherwise;

nullify: corresponding foreign key attributes are set to null (assuming that this value is allowed by other constraints); or

cascade: the delete/update operation is applied to tuples with matching foreign key, and if necessary repeated at lower levels.

(Section 8.2 includes further comment on these suggestions, with particular reference to the similarities with the network approach which they contain.)

In one approach all possible affected relations and tuples must be checked first to ensure that no action is subject to a restrict option, in which case the original operation is rejected. Alternatively, a 'snapshot' of the database may be used to allow processing to continue on the assumption that it is valid, and if it prove invalid this snapshot then allows the database to be restored to its previous (and presumably consistent) state.

Thus specifying a foreign key requires explicitly stating

- the attribute list and target relation,
- whether null values are allowed,
- the appropriate action for update operations, and
- the appropriate action for delete operations.

Whether or not null values are allowed in a foreign key is not just a case of enforcing rules relating to the integrity of the database; it is a semantic property of the data-model, and depends on the designer's understanding of the part of the real world the model represents.

Suppose the example database is extended to include a relation

CITY(<u>CITY</u>, details)

and that the CITY attributes of the SUPPLIER and JOB rela-

tions are foreign keys referencing the CITY relation. Can there be a job unless it is known in which city it is located? Probably not. But it may well be appropriate to record some details about a new supplier in the database before it is decided which of several available locations will be most appropriate. In this case it might be decided to allow nulls for the foreign key CITY in the SUPPLIER relation, while disallowing them for the same attribute in relation JOB.

While the integrity of the database can be maintained automatically through the careful selection and specification of foreign keys, they are in themselves a move towards explicitly stating the major inter-relationships between data sets, i.e. they compromise the initial principles on which the relational approach was founded.

(The argument presented here for primary and foreign keys has drawn mainly on the two concepts of integrity given at the beginning of this section. For completeness it should be pointed out that a third form of integrity exists, the ability of the user to define the acceptable values, or range(s) of values, for each attribute or domain. This form is not relevant to the present discussion.)

2.4.3. Assignment

Suppose now that A and B are two real relations which are union-compatible, that is, the corresponding attributes of each share common domains; it becomes necessary to answer a number of questions about even quite simple operations

such as

A UNION B

While the domain set of the result is that of either A or B, what of the attribute set? Are the names taken from A or B? This may be resolved by the definition of the UNION operator.

And supposing that a1 is designated the primary key of A, with a2 as an alternate key, while for B the primary key is b2, what can be said of the keys of the union? If there already exists in the database a real relation C union-compatible with A and B (and which in fact may be either of them) the assignment

C := A UNION B

solves these problems, with C's existing tuple set being replaced by the result of the union operation. The definitions in the previous section imply that an assignment is only possible when, as in this case, a suitable real relation is available, i.e. the assignment operator cannot create a new real relation. (One should not be able in any case to assign to a virtual relation.)

But this does not remove all the problems if the DBMS allows null values. Since C already exists as a real relation, its primary key, and possibly any alternate keys, are already known. No tuple may have a null value for (any attribute of) its primary key. Now if c1 is designated as C's primary key, the definitions given above

for A and B make it possible that B-tuples may exist with null value for attribute b1, and these tuples will necessarily all appear in the result of the union, but are not allowed to appear in C. Thus during replacement of the tuple set required by the assignment operator, such tuples must be eliminated.

A further consequence of the restriction on creation of relations is that should it be necessary during the course of data manipulation to hold the value of an intermediate result for later use, then either

- a real relation must be explicitly created for this purpose, or
- a virtual relation must be defined as the result of the expression leading to the required intermediate result.

The first of these is clumsy, and the second does not really answer the need.

Since virtual relations are dynamic, in the sense that they are not physically stored but are retrieved by evaluating the defining relational expression at run-time, using the current values of any real relations referenced, the user must be aware that in the absence of a locking mechanism to inhibit updates on any of these real relations, two consecutive references to the same virtual relation may produce different results. (Some systems have introduced the notion of a transaction, and inhibit

re-evaluation of virtual relations more than once in their scope. This allows a concurrent user to proceed, possibly updating one or more of the base relations, with neither being aware of the activity of the other.)

It is argued that these restrictions may be used to enforce desired constraints on the database, since creation of real and/or virtual relations may be restricted to someone authorised to do so, such as the Database Administrator; but it also inhibits the development of solutions to ad hoc queries unless the required expression may immediately be written as a single expression.

The solution adopted in MURDER is to allow assignment to create a named relation in the user's workspace. This relation may be used in subsequent expressions, or moved (possibly after further manipulation) to a real relation, but is not considered to have permanent existence, i.e. it will not be saved when the database is closed. (This approach is borrowed from DSL-ALPHA, which is described briefly in Chapter 6.)

If the syntax for creation of a real relation were altered to

```
CREATE RELATION relation_name
                (attribute_domain_list)
              [= relational_expression]
       PRIMARY KEY ...
```

assignment would become possible. It is not possible to extend the syntax of a relational expression to allow assignment unless an attribute-aliasing mechanism is

provided.

2.4.4. Nulls

The previous section has already introduced some of the problems created by allowing a database to contain null attribute values.

In general the theory of nulls in relational systems is only imperfectly understood, and it is perhaps better - and certainly easier - to play safe by avoiding them, as MURDER does. (Date argues that this is the best approach at present, suggesting that they are more trouble than they are worth, being primarily a rich source of confusion and error. Date[82, 87a]) What follows is only a brief indication of some of the inherent problems.

Firstly, what is meant by a null value? Intuitively it is merely an indication that a value is unknown, either because a choice between several possible values has not yet been made, as in the supplier-city example above, or because as yet only incomplete information is available. (The user may in fact be unaware of this, for example if his view of a relation excludes some attributes. This is one reason for the apparently arbitrary rules about updating real relations through views which are imposed by some DBMSs.) But it may also indicate that an attribute is in some sense not applicable to a given tuple.

If relation SPJQ records the information that a certain supplier is to supply a certain part to a certain job in

some as yet undetermined quantity, the tuple (S1, P2, J3, ?) might be included to indicate this situation. Now a function like

AVG(SPJQ.QTY)

must ignore that null value if it is not to return the value '?', although ideally it might warn the user that this has occurred. If COUNT(SPJQ) counts the tuples in relation SPJQ,

AVG(SPJQ.QTY) ≠ SUM(SPJQ.QTY) / COUNT(SPJQ)

which might cause unexpected results. Even the COUNT function is suspect, as is shown below.

In the comparison

a θ b

where a and b represent values from a common domain, and θ represents a relational operator, the result is defined as 'unknown' if either (or both) of a and b is null. The logical operators and (&), or (|) and not (^), also now require expanded definitions using a three-valued logic, to take account of this new situation where a comparison may be true (T), false (F) or unknown (?):

&	T	F	?
T	T	F	?
F	F	F	F
?	?	F	?

|	T	F	?
T	T	T	T
F	T	F	?
?	T	?	?

^	
T	F
F	T
?	?

Next consider the 'set'

A = {a, b, c, ?}

Does the '?' indicate 'completely unknown', or (as is perhaps implicit in the above equality) 'unknown, but known to be different from all of a, b and c'? With the former definition, the best the function COUNT(A) can do is return 'either 3 or 4', while the latter definition allows COUNT(A) to return 4 unambiguously. However this is not necessarily better, since now the DBMS has to be able to deal with any number of different nulls! Consider

B = {a, b, c, ?, ?}

Does this have 4 or 5 elements? Is it even valid?

What about set membership? Both predicates

d ∈ A

? ∈ A

must evaluate to 'unknown'.

In fact almost all the familiar concepts, as well as the predicates and operators, need expanded definition to cope with the problem of null values. Some work has already been done in this direction, but it remains a difficult area. (Date[83b])

2.4.5. Indexes

Indexes are really not part of a relational DBMS at all, but this brief discussion is included because indexes,

either implicit or explicit, seem to be a necessary adjunct to providing acceptable response speed. An index provides rapid access to tuples in a relation, based on the values of one or more attributes.

Many operations on relations involve identifying one or more tuples which satisfy some specified condition. This is a potentially costly operation in terms both of time and resources. An index may help by providing relatively fast access to tuples satisfying all, or part of, the search condition - just as the index to a book allows the reader to find which pages contain material relevant to his topic of interest. Indeed some queries might be answered entirely within the indexes and without reference to the data pages at all.

Yet such an index should not be 'visible', otherwise one of the basic criteria of a relational system - that the relation be the only data structure - would be violated. The DBMS could index each attribute of each relation without the user's knowledge, but this could prove costly (especially if some of the attributes are never - or only rarely - used in search conditions), because of the maintenance of appropriate indexes whenever a tuple is added, deleted or modified.

Although discussion of SQL is deferred until Chapter 6, it is introduced here to illustrate one approach to this problem. The compromise adopted in SQL is to allow the user to indicate that an attribute is commonly used in

searching by explicitly creating an index, but not allowing other reference. If the DBMS can take advantage of this index in evaluating a query, it may, but the user does not control whether or not the index is so used.

However this use is blurred by having considerations of integrity overlap considerations of efficiency. Thus it is possible to

```
CREATE [UNIQUE] INDEX index_name
    ON real_relation attribute_name...
```

The optional keyword UNIQUE (or DISTINCT in some implementations) determines whether more than one tuple may have the same value for the named attribute(s). Obviously a unique index should be created for the primary key, it should be created before any tuples are added to the relation (otherwise the CREATE INDEX command could fail), and it should never be deleted by a

```
DROP INDEX index_name
```

command. (To do so would be to lose control of the integrity of the database. Because of the cost of maintaining indexes, IBM[84] recommends that, whenever significant changes are to be made to a relation, the index should be deleted before the changes are made, and recreated afterwards!) Note, however, that there is no means of identifying the primary key as such, when more than one unique index exists for a relation.

Specifying the primary key at the time of creation of each real relation avoids this problem, possibly also providing one index into the relation. (The overlap this way is acceptable, since there is no requirement to use the primary key as an access mechanism; in the previous approach the index was the only means of specifying required integrity constraints.)

CHAPTER 3

MURDER

This chapter describes the basic VAX 11/750 implementation of MURDER, and compares it with the theory set out in Section 2.3.

3.1. Overview of Implementation

As indicated earlier, a relation consists of a header, together with a possibly empty set of tuples. The header must give - at least - the details of attributes in order, so that the stored representation of the tuples may be interpreted correctly. Although the order of attributes may not be significant, it must be consistent for this interpretation to be possible.

A relation in MURDER is stored in a linked list of pages, as keys are stored in the lowest level of a B+ tree. The size of a page is the size of a disk transfer unit, 4K bytes. The structure of a page is shown in Figure 3.1. This effectively imposes a maximum size of 4088 bytes on a

---

```
struct page
{  char data[4096 - sizeof(int) - sizeof(ptr)];
   int population;              /* # of tuples in page */
   struct page *nextpage;     /* pointer to next page */
} page, *pageptr;
```

Figure 3.1 - Page Structure

---

tuple. Tuples from more than one relation are not mixed in a page, and within a page tuples are maintained in strictly increasing order (with character and numeric data interpreted appropriately). MURDER insists that the primary key be the first attribute(s) in a relation. This ordering allows the relation itself to act as the primary key index.

As a consequence of this design decision, most algebraic operations (together with building the universe in a calculus query, described in Chapter 5) are inherently simplified. While top-to-bottom order within a relation is not significant, there must be some order, even if it is only the physical storage order within the database. This sorted 'system' order is used as the basis for efficient implementation of the algebra and calculus described in the following chapters.

When tuples are added to a relation, either singly or during a projection other than over the first attribute(s), the appropriate page for each new tuple is found by traversing the linked list of pages examining the first and/or last tuple in each. (For most primitive operations, which generate relations in 'sorted' order, new tuples are simply added after any existing tuples.) The first page to be examined is the current page, if any, as shown in Algorithm 3.1.

```
if there is no current page
then if the relation is empty
     then get new page
     else get first page
          scan
else if new tuple < first tuple in current page
     then get first page
          scan
     else scan


while new tuple > last tuple in current page
  and there is a next page
   get next page
```

Algorithm 3.1 - Basic Find and Scan Procedures

The scan procedure leaves the current page pointer set to the appropriate page. A binary search of the tuples in the page then determines whether the tuple already exists in the relation, and if not where it is to be placed. If space exists in the page the new tuple is inserted (with higher tuples being moved up if necessary); if not, a new page is inserted into the linked list, with the old page being split across the two pages at the insertion point. There is no attempt to balance tuples across the two pages, although when the database is opened the pages of each relation are packed. Note also that when a relation is created in sorted order - as is usually the case - this algorithm automatically fills pages.

The same lookup procedure is used to find a tuple which is to be deleted. Any additional processing required by the existence of foreign keys is discussed later.

Figure 3.2 gives the stucture for each attribute in the header. The parent relation is included to allow for aliasing, described in Chapter 4.

Externally, MURDER supports three data types:

fixed length character string,

decimal (like COBOL S9(7)V99), and

integer

There are three predefined domains,

```
DATE    : char 8 (dd/mm/yy)
DECIMAL : decimal
INTEGER : integer
```

Further domains are created as required by

```
CREATE DOMAIN domain_name type

type ::= C length |
         D |
         I
```

```
struct attspec
{  char pname[NAMESIZE],          /* parent relation */
        aname[NAMESIZE];          /* size = 12 bytes */
   int adnum,                  /* index to domain table */
       atype,           /* 0,1,2 for char,decimal,int*/
       asize,                          /* in bytes */
       aoffset;                /* from front of tuple */
} attspec, *attptr;
```

Figure 3.2 - Attribute structure

Domains may be discarded (provided that no attribute is still based on them) by

DROP DOMAIN domain_name [domain_name] ...

The system maintains a table of current domains, giving for each its name, number, type and size. (See Figure 3.3.) New domains are added at the end. Deleting domains requires the table to be updated, but this is not likely to be a common processing requirement.

There is no realization of the order flag described in Chapter 2. All domains are considered to be ordered, using algebraic value and lexicographic order for numeric and character data respectively. In the early version attributes were considered to be compatible if they were the same type and size; now they simply need to be based on the same domain. (This is still less rigorous than the theory requires, but is workable. McLeod[76] provides a full discussion of this problem, which was introduced informally in Chapter 2.)

```
struct domain
{  char dname[NAMESIZE];
   int dnum,
       dtype,
       dsize;
} domain, *domptr;
```

Figure 3.3 - Domain Structure

Internally, both decimal and integer data types are represented as floating-point numbers. This limits the size of integers which can be represented exactly, but simplifies processing, particularly when arithmetic functions are involved. A COUNT always has domain INTEGER, and the others - AVG, SUM, MAX and MIN - have the same domain as their argument. Only AVG is unusual in this respect, as the average of a set of integers might be expected to be a real value. Storing integers as floats removes this problem, e.g. in a condition like

```
WEIGHT > AVG(WEIGHT)
```

With this introduction the structure of a relation node is given in Figure 3.4. (See also Figure 3.5.) Note that

```
struct relnode
{  char relname[NAMESIZE];
   int numatt,        /* # of attributes in relation */
       relsize,           /* size of tuple in bytes */
       numkey,        /* # attributes in primary key */
       numfk,           /* # foreign keys in a tuple */
       numref,           /* # referencing relations */
       maxt,            /* max # of tuples per page */
       numtup;           /* # of tuples in relation */
   struct relnode *prevrel,   /* doubly linked list */
                  *nextrel;    /* of relation nodes */
   attptr header;    /* to array[numatt] of attspec */
   pageptr firstpageptr;      /* to first data page */
   refptr fktab,         /* to arrays giving details */
          reftab;           /* of related relations */
} relnode, *relptr;
```

Figure 3.4 - Relation Structure

relsize = sum of attribute sizes + sizeof(int)

The use made of the extra integer, along with the representation and handling of foreign keys, is given in Section 3.2.

Keeping the tuple count in the relation node serves two purposes. Evaluation of a COUNT function is now trivial, but more significantly the operations of opening and closing the database are much simplified. Relations are written in order - first the node (excluding any pointers), then the header, details of any references, and finally

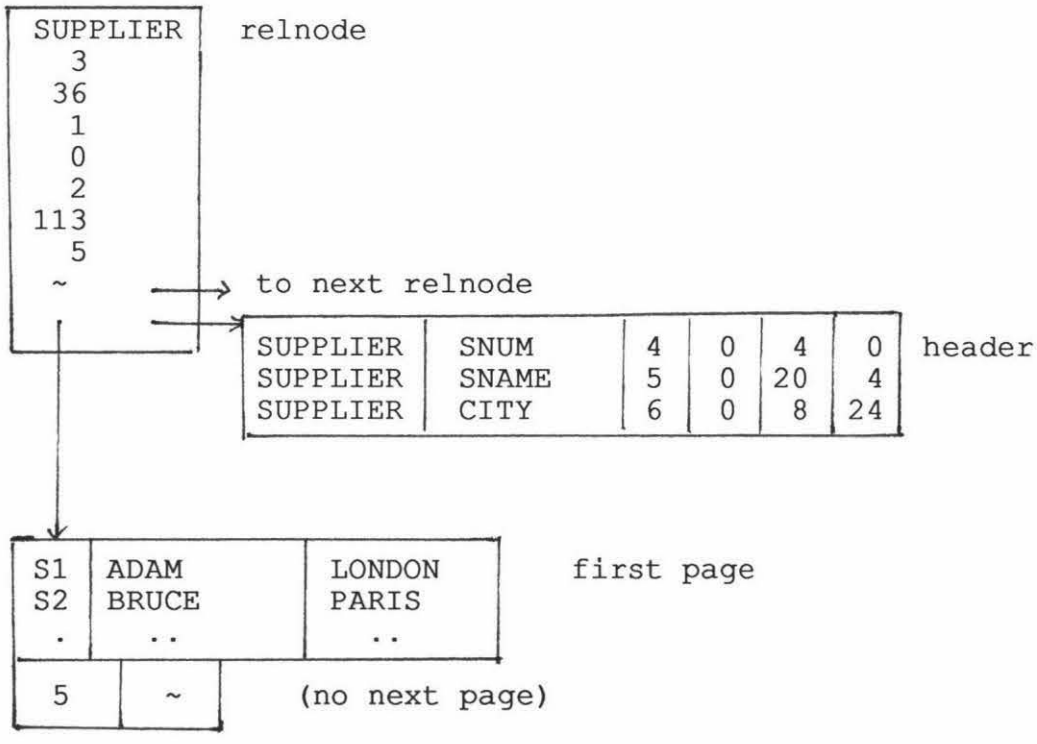


Figure 3.5 - Basic Representation of SUPPLIER relation (reference tables omitted)

the data tuples. Since each page keeps a population count, this is trivial. On opening, once the relation details have been read, all the information needed to rebuild the relation is available. The maxt field is only a convenience, to avoid recalculation whenever a tuple is to be added to a page, but here it allows data pages to be packed simply while opening the database. It will be apparent that the database can thus be saved in a simple stream file.

Permanent, temporary (or workspace) and transient relations are maintained in three doubly-linked lists. Although insertion is always at the tail (to simplify maintaining foreign key reference details), the double linkage facilitates deletion, which may occur anywhere.

The earlier file-oriented version required slightly more complicated treatment. At present, the memory-resident version is suitable only for relatively small data volumes, but as the cost of memory continues to decrease, the use of memory-resident databases will increase.

3.2. Relation Definition

Relations are created either explicitly or as the result of an assignment. The first of these, used for permanent relations, is

```
CREATE RELATION rel_name
          ( att_name : domain_name
        [ , att_name : domain_name ] ... )
      PRIMARY KEY att_name [ , att_name ] ...
  [ ; FOREIGN KEY att_name [ , att_name ] ...
                REFERENCES rel_name ] ...
```

This creates an empty relation. (It is required that no relation of the same name already exists). Attribute names must be unique within a relation, and domains referenced must already exist.

The primary key is required to be the first attribute(s) specified, which allows the relation to act as its own primary index. Any relations referenced must also already exist - this restriction means that recursive or cyclic representations are not possible, but this is not likely to cause problems. Additionally it is required that when a foreign key has more then one attribute these attributes occur consecutively in the referencing relation. Here there might be a problem in cases where a sequence of attributes contains references to a number of other relations, e.g. if the attribute-pair SNUM, JNUM in relation SPJQ is to be a foreign key to a new relation SJ.

References are maintained using the structure of Figure 3.6. Each permanent relation maintains separate tables of these structures. The relnum field is used to rebuild the

---

```
struct refspec
{  relptr refrel;     /* pointer to related relation */
   int relnum,      /* of related permanent relation */
       ranum,     /* first attribute of foreign key */
       rnumatt;      /* # attributes in foreign key */
} refspec, *refptr;
```

Figure 3.6 - Reference Structure

---

representation of all associations when the database is opened, since the list of real or permanent relations is ordered, with new relations being added at the tail. For each such relationship, both the referenced and referencing relations maintain these details, although the last field is only used in the latter case. This involves some duplication, but processing is simplified, and as creating or discarding a permanent relation occurs only infrequently, overheads are not a factor.

Individual tuples may be added to, or removed from, permanent relations by

```
INSERT rel_name < value_list >
DELETE rel_name < value_list >
```

where

```
value_list ::= value [ , value_list ]
```

For insertion, values for each attribute must be given; for deletion only the primary key values are required. Of course, each value must be of the correct data type.

When a tuple is to be inserted, the table of foreign keys is checked. For each referenced relation a lookup, based on the foreign key in the candidate tuple, is executed. When the tuple is found, the extra integer allowed for in relsize is incremented. Finally, the new tuple is inserted in the appropriate position, with this integer, the reference count, set to zero, and the page population and relation tuple count are both incremented. If any

lookup fails to find a tuple matching the foreign key the insertion is rejected.

The findtuple function which performs the lookup has three parameters - pointers to the relation and tuple, and an integer indicating any action to be taken:

'+'	insert, assuming not found
'-'	delete, assuming found and reference count zero
-1, 1	decrement or increment reference count
0	check if reference count is zero

The function returns true or false as appropriate. (Note that in C a character is compatible with an integer, so that the apparent mismatch of types shown above is legal. If the relation being searched has its numkey field greater than zero, only that number of attributes are checked during the lookup; otherwise - a temporary or transient relation - all attributes are considered.)

Deletion of a tuple first checks that the reference count is zero, refusing the operation otherwise. (This corresponds to the restrict option suggested in Date[86], and outlined in Section 2.4.2.) Then similar lookups, which should be successful unless database integrity has been lost, are executed, decrementing the reference count in each case. Finally the page population and relation tuple count are also decremented.

New pages are acquired, and empty pages discarded, as required.

Higher level manipulation is also provided:

```
INSERT rel_name ws_name | argument

argument ::= alg_expr |
             calc_expr |
             sql_expr
```

will attempt to insert each tuple from the named workspace relation, or the result of evaluating the argument expression, as appropriate. The operation will fail if any tuple already exists in the permanent relation. Suppose that wi, rj are equal tuples of relations W, R respectively, and that rj has a non-zero reference count. Inserting wi into R must not change rj's reference count, otherwise the referential integrity of the database will be lost; and if the insertion is not flagged as an error, a subsequent attempt to delete wi - an operation which might reasonably be expected to succeed - must fail precisely because this count is positive. While it might be argued that it is sufficient to reject only w-tuples already in R, or even only such tuples where the count is non-zero, a little reflection should show that the above rule, rejecting the whole operation, is not likely to be a problem. For example the sequence

```
X := INTER R WITH W
INSERT R DIFF W
         WITH X
```

identifies the offending tuple(s) and inserts the rest.

```
DELETE rel_name [ ws_name | argument ]
```

attempts to delete all tuples, or those in the temporary

relation or expresssion result, from the permanent relation. In the former case, all tuples in this relation must have zero reference counts; for the other cases all tuples from the temporary relation must be present, and must similarly all be unreferenced.

The second method for creating a relation applies to a temporary relation in the workspace. This is by assignment:

assignment ::= rel_name := argument

If the relation on the left already exists in the workspace, it is deleted immediately before the assignment occurs, i.e. it may be referenced in the expression on the right. (In practice, an expression results in a relation in the third, transient list, which is also used for intermediate results, and the actual assignment is accomplished by transferring the relnode to the temporary list, after deleting a node of the same name if required. Any relations remaining in this transient list are deleted at the end of execution of each statement.) Although tuples in a temporary relation are unique, such a relation has neither primary nor foreign keys. The concatenation of all attributes constitutes a key, although it may not satisfy the minimality requirement for a candidate key. Having all expressions evaluated in the temporary list, with results explicitly transferred to permanent relations by INSERT or DELETE statements, has the effect of allowing expression evaluation to proceed without interruption by

foreign key considerations, thus keeping the overheads of integrity constraints to a minimum.

Note that there is no simple assignment like

```
A := B
```

since none of the arguments given above is simply a relation name. However, if A is a compatible empty permanent relation,

```
INSERT A B
```

achieves the same effect as assignment.

In an assignment the new relation has already had all its details filled in during the expression evaluation. Attribute names and domains are inherited from the operand relations in a well-defined manner. If these names are not adequate (e.g. because of the duplication of an attribute name) they can be qualified by the name of the parent relation, or by an alias if appropriate.

Algebra, calculus and SQL expressions are described in Chapters 4, 5 and 7 respectively.

The last statement used in data definition is

```
DROP RELATION rel_name [ rel_name ] ...
```

This removes all reference to the named permanent relation(s), provided that each is not referenced by any other relation (which implies that any tuples remaining in the relation are unreferenced). Analagously, there is

also

DROP WORKSPACE [ws_name [ws_name] ...]

to discard all, one, or several temporary relations.

This brief overview should provide an adequate background for a discussion of the algebraic operations, which is given in the next chapter.

3.3. Additional Features

For completeness this section gives an overview of other MURDER facilities, which fall into two main categories:

utilities, and
experimental facilities.

3.3.1. Utility Statements

These serve mainly to allow the user to manage the database.

OPEN db_name

must be executed before any other processing. If no database file 'db_name.db' is found, a new database is initialised.

CLOSE

will exit the database, after first prompting the user to check whether any changes made to permanent structures should be saved. The user may then open another database.

Q | QUIT

will close any open database as above, and exit the program.

LIST rel_name | argument

attempts to find a named relation first in the permanent list, then in the workspace, and finally attempts to evaluate the expression. The relation (or expression result) is displayed at the user's terminal, screen by screen, prompting the user before beginning a new screen. Each screen begins with the header information, one line for attribute parent names (only if not all have the same parent), one line for the attribute names, and these are then underlined, using '=' if the attribute is part of the primary key and '-' otherwise.

Simple data dictionary facilities are provided by two statements, WHAT and WHERE.

WHERE domain_name

lists all relations having an attribute based on the named domain.

WHAT DB

list all environment constants (explained in the next section), together with the current value of each.

WHAT DOMAIN

lists all domains with their data type.

```
WHAT RELATION [ rel_name ]
```

either lists all permanent relations (unqualified case), or lists the attribute-name, domain-name pairs for the named relation. Primary key attributes are marked as such by an asterisk.

```
WHAT WORKSPACE [ rel_name ]
```

supplies similar information for temporary relations. There is a final form

```
WHAT MACRO [ macro_name ]
```

which either lists available macros (unqualified form) or gives the expansion of the named macro. (Macros are also explained in the next section.)

3.3.2. Experimental Statements

These were designed as a means of understanding the problems involved in making a relational system directly usable for data processing, rather than merely a vehicle for data storage and retrieval.

There is actually a second way of creating a temporary relation, using

```
MAKE rel_name FROM argument SET assignment_list
assignment_list ::= assignment [ ; assignment_list ]
assignment ::= att_name [ : domain_name ] = expr
expr ::= term [ addop expr ]
term ::= factor [ mulop term ]
```

```
factor ::= att_id |
           env_const |
           literal |
           ( expr )

addop ::= + |
          -

mulop ::= * |
          /

env_const ::= DB . att_id
```

The domain must be specified in an assignment unless the expression consists only of an attribute or an environment constant (in which cases the domain can be inherited). Any att_id referenced on the right of an assignment must refer to an attribute in the argument, or to an attribute defined on the left of an earlier assignment in the list. (Section 4.1 elaborates the definition of att_id in a more helpful context than is available here; for the moment it may be regarded as a synonym for att_name.)

The special relation DB is predefined and always contains exactly one tuple. It has three predefined DATE attributes (respectively the current date, and the dates the database was created and last updated). Further attributes can be created, changed or discarded by

```
DECLARE att_name : domain_name [ = value ]
LET att_name = value
FREE att_name
```

This is in fact the only part of MURDER which allows facilities comparable to adding new attributes to an existing relation, or removing existing attributes; and it

is also the only place where nulls appear, although in a weak form as the default value to new environment constants, viz. spaces for character data, zero for numeric.

A real relation may be updated by the MODIFY statement:

MODIFY rel_name [WHERE sel_expr] SET assignment_list

The selection expression (described in Section 4.2) identifies tuples to be changed, in which the assignments are applied; tuples not satisfying the expression are left unaltered. If no selection is specified all tuples are affected. Since no new attributes are being created, all assignments have the form

att_name = expr

There are some additional constraints required by considerations of primary and foreign keys, and if these are not met the whole operation is rejected.

(1) A non-key attribute may always be modified by any appropriate expression.

(2) If a primary key attribute is to be modified the selection must only reference primary key attributes.

(3) When a key attribute (primary or foreign) is modified, the expression must consist of a single constant.

(4) A foreign key attribute, not part of the primary key, may only be altered if the modified tuple still

references a tuple in the referenced relation.

(5) If a foreign key attribute is part of the primary key, the conditions of Rule 4 apply; additionally the change is propagated downwards through referencing relations.

(6) A primary key attribute not part of a foreign key may only be modified to a value which will not cause duplication of a primary key. This will also be propagated through referencing relations if required.

These rules have been found to be reasonable in practice, and correspond to the cascade option of Date[86], outlined in Section 2.4.2. Rules 2 and 3 are basically common sense, and ensure that no 'side effects' occur. Rules 4 and 5 ensure that referential integrity is maintained, and Rule 6 enforces entity integrity and prevents the loss of data which would occur if tuples are allowed to 'merge' at this 'top' level.

Tuples are, however, allowed to merge at lower levels. This requires careful treatment of reference counts:

(1) No keys are affected, and hence no reference counts are changed.

(4) References from the original tuple are deleted, and replaced by references from the modified tuple. (Since no primary key is affected merging cannot occur.)

(5) Processing is as in case 4, but if merging occurs references from the duplicate tuple are removed, and the reference count of the surviving tuple is the sum of the counts from the duplicates. (This reflects consequent merging which may occur in referencing relations, when references from duplicates are removed as above.)

(6) Since in this case no merging can occur, processing is straightforward.

When a primary key is modified the change may cascade through referencing relations. However, since the selection expression in this case may only reference primary key attributes, and since these are the first attributes in the relation, the expression representation needs only simple modification to apply at lower levels, viz. all attribute offsets are increased by the offset of the foreign key in the referencing relation. The assignment list representation is first purged of assignments to other than primary key attributes before the result offset is similarly increased. (Note that this description is of the original implementation; modifications, described in Chapter 7, were required to implement the SQL UPDATE statement.)

The MAKE statement was a first attempt at what might be described as 'row' processing, corresponding to record processing in a file environment. The basic unit of a relational system, the relation, is too coarse for many

practical purposes. Since much data processing involves associating a row from one relation with a group of rows from another, MAKE went some distance towards satisfying processing requirements, but still proved inadequate. (It will however be the basis for extending the SQL implementation to include expressions in the select list, described in Section 7.1.) MODIFY is in this sense also a row-processing statement.

The complementary 'column' processing is provided by a group of functions:

```
fn_stmt ::= COUNT ( fn_arg ) |
            function ( fn_arg . att_id )

fn_arg ::= rel_name |
           [ argument ]

function ::= AVG |
             SUM |
             MAX |
             MIN
```

MURDER initially provided a macro facility for convenience during testing. A macro is stored like a relation with one unnamed attribute of type C 76 (a 76-character string), and contains implicitly numbered MURDER command lines. Since a macro may not extend beyond one page the maximum length is 53 lines.

Macros are defined, changed and deleted, by sub-commands within the EDIT statement. Execution of a macro simply requires reference to its name. A macro may accept parameters, and may execute calls on other macros, although recursion is not supported.

This facility was recognised as useful for processing (the stored program concept?), and two additional statements - FOR and IF - were borrowed from normal programming languages. The first is

```
for_stmt ::= FOR EACH rel_name . att_name stmt_seq ENDFOR
stmt_seq ::= stmt [ stmt_seq ]
```

which executes a loop with the control variable taking each possible value of the attribute in turn. Within the statement sequence any statement may appear, and the symbol '$' may be used to refer to the current value of the control variable. Thus there might be a macro

```
FOR EACH SUPPLIER.SNUM
    LIST_PART($)
ENDFOR
```

with macro LIST_PART defined as

```
%1
LIST SEL SP WHERE SNUM = %1
```

The notation '%1', '%2' ... refers to the first, second ... parameters. The first line above is an example of a literal statement, which is simply echoed at the user's terminal, while the second statement lists SP-tuples for the supplier identified by the parameter value.

This statement needs to be extended to allow for iteration based on multiple attribute combinations, or at the least the primary key in the named relation, but this has not been done because of the complications which would result from reference to such a multiple loop control variable

within the scope of the FOR statement - but see the discussion in Sections 4.7 and 8.2, where it is urged that support for referential integrity should be recognised by forms of expressions which treat a key as a single entity.

For the second new statement MURDER has

```
if_stmt ::= IF [ NOT ] EMPTY rel_name stmt_seq
               [ ELSE stmt_seq ]
            ENDIF
```

rather like an option commonly provided in network database systems to test whether an instance of a record-type owns, or is a member of, an instance of a named set-type. As for IF statements in common programming languages, execution of either statement sequence is controlled by the condition value.

There is also a rudimentary report generator, loosely based on COBOL's Report Writer module:

```
report_stmt ::= REPORT rel_name [ option ] report_group

option ::= PRINT |
           DISPLAY

report_group ::= header_group report_group footer_group |
                 detail_group

header_group ::= HEADING [ FINAL ] line_group

footer_group ::= FOOTING line_group

detail_group ::= DETAIL line_group

line_group ::= line_spec [ ; line_group ]

line_spec ::= LINE [ + ] integer [ field_group ]

field_group ::= field_spec [ , field_group ]

field_spec ::= COL integer data_spec
```

```
data_spec ::= att_name |
              env_const |
              literal |
              SUM ( att_name )
```

i.e. the report may be sent to a line-printer (line length 132 characters) or to a VDU (the default, line length 80 characters), and consists of matching heading/footing groups surrounding a detail group, or just a detail group.

If FINAL is used it must appear in the first heading, and then specifies that there is to be an overall heading and footing for the report. Other headings and footings are generated when changes occur in the value of the appropriate control variable, i.e. the first, second ... attribute in the named relation, as in normal control-break processing. (Again an extension could allow for control-breaks on multiple attribute values.)

Each group may be one or more lines, each line may contain zero or more fields, and each field may be an attribute in the named relation or the special relation DB, a literal, or - in the case of a footing only - the sum of a numeric field referenced in the detail group. More than one footing may sum the same attribute. Each sub-total is initially zero, and is reset to zero after output and, if necessary, adding it to the next. Lines may be positioned relative to the current line (the form '+ integer') or absolutely; and in the latter case a page-throw may be generated before the line is output. (For a report sent to the user's terminal the user is prompted for a carriage-return before clearing the screen and

continuing.)

These experimental statements were designed as a means of coming to an understanding of relational processing. In particular macros are not efficient, because the stored text is re-interpreted on each call, and the parameter mechanism is simple text-substitution, but they did make it clear why there is such emphasis on embedding DBMS statements in a host program.

In one other feature MURDER adopts its own experimental approach. Two databases may be open simultaneously, which may be called the Host and Data databases. While the former may only contain macros, the latter will normally - but need not - contain only relations. Thus the same macros may be applied to more than one data-set, or different groups of macros may be applied to the same data-set (which proved useful in assessing student work).

On the VAX 11/750s the MURDER system may either stand alone, interacting directly with the user, or may be 'driven' by a host program sitting between the user and the system. (For this two UNIX pipes are used.) The host program is thus essentially a user-oriented menu/data-input system, augmented by processing beyond the direct capabilities of MURDER.

Also implemented on the Prime 750 is a library of subroutines called ATM (for Acessory to Murder!) which may be bound to a host program and which provides limited record-at-a-time processing: macros may be executed, or

relations read in first, next order, and tuples added, changed or deleted.

3.4. Evaluation of MURDER

How well does MURDER measure up to the standards set out in Chapter 2?

Certainly the only visible data-structure is the relation - it is in fact the only data-structure, if the macro facilities are excluded; the special relation DB is a relation, although it is manipulated by separate statements.

Domains are not fully implemented, but there is enough to be useful. Processing was actually simplified by the inclusion of domains. Perhaps the range of data types could be widened.

The order of attributes in a relation in MURDER is fixed when the relation is created, either explicitly or by assignment. While this order has no inherent significance, many primitive operations depend upon it. In particular, it is often tacitly assumed that the primary key of a temporary relation W is the first n attributes, for some n, $1 \leq n \leq a(W)$, the arity or degree of W.

Further, the top-to-bottom order of tuples in a relation is strictly sorted according to the (current) attribute ordering, and considerable advantage is gained by this in obtaining an efficient implementation of the primitive operations. (This point is further expanded in Chapter 4,

when the algebraic primitives are described.) If U and V are two compatible tuples

U = (u1, u2 ... un), V = (v1, v2 ... vn)

the predicate

U < V

is true if and only if

ui < vi, for some i, $1 \leq i \leq n$
and uk = vk, for all k, $1 \leq k < i$

with other comparisons similarly defined.

On the subject of keys MURDER is slightly deficient, in requiring the primary key to come first in a relation, and foreign keys to be consecutive attributes. However what has been included was achieved simply, so simply in fact that it is surprising that they are not supported in commercial DBMSs. The distinction between permanent and other relations contributes much to this.

Views are not supported, although the macro facility could easily be extended to do this. A macro consisting of a single expression would be sufficient. This would also simplify the scope problem briefly mentioned in Section 2.4.3 - a view would persist either for the duration of the macro which caused it to be generated, or for a single statement, as appropriate.

The separation of permanent and temporary relations allows meaningful assignment without the problems caused by integrity constraints. (As an aside, Chamberlin[76], which introduced the language SEQUEL2, provided a similar assignment, although additionally the attributes could be named; but in all dialects of SQL the key concept is indistinct - see the discussion of this, and also Alpha's workspace concept, in Chapter 6.)

The absence of nulls - and a recognisable default value is not really a null - is a major omission, but one which is supported by writers such as C.J. Date. (Date[87a])

Since MURDER is a single-user system, problems of concurrent usage are not considered. Several users may open the same database, but each is automatically given his/her own copy in their local directory.

Codd[85] suggests a simple rating scheme for measuring how fully a RDBMS complies with currently perceived requirements. He lists a number of rules - 12 basic, 9 structural, 18 manipulative and 3 integrity - which are necessary in a faithful implementation of the relational model, and evaluates three commercially available systems for their compliance with these rules. His results:

IBM's DB2	46%
IDMS/R	8%
Datacom/DB	10%

On this measure, MURDER achieves better than 40%, making

up for DB2's treatment of null values by the inclusion of domains and (entity and referential) integrity. Since these have been shown to impinge only slightly on other processing, it is indeed surprising that more systems have not seen fit to implement them. Of the three systems evaluated by Codd, none supports them to any usable extent, although DB2 does include a type of primary key facility, using the concept of separate indexes provided by SQL.

CHAPTER 4

MURDER Algebra

This chapter describes the implementation of the eight commonly available algebraic operators. These are all a necessary background for the relational calculus to be described in Chapter 5, but also highlight the advantage to be gained through ordering the physical storage of tuples.

An algebraic expression may appear as the argument to a utility statement, or as the right-hand side of an assignment to create a temporary relation:

```
rel_name := alg_expr
```

If the result relation already exists in the temporary list, it is deleted immediately before the assignment. All algebraic expressions return a (pointer to a) relation, and involve one or two instances of alg_primary, defined as

```
alg_primary ::= rel_name [ = alias ] |
                '[' alg_expr ']'
```

Here any relations named must already exist. The brackets '[' and ']' indicate recursion, i.e. an operation may act on the relation returned by a previous operation. (Recall the use of single quotes in syntax diagrams to denote an instance of the quoted character - see Section 1.3) When two instances of the same relation are involved, at least one of them must have an alias if the user wishes to

distinguish between duplicate qualified names.

Algebraic operators may be classified either as unary or binary (one or two operands respectively), or alternatively as primitive or derived (the latter being operations defined in terms of primitive operators but which are included for convenience because of frequency of use).

The unary operators supported by MURDER are

> projection - a 'vertical' subset of a relation, not necessarily retaining all attributes of the operand, and possibly re-ordering those retained; and
>
> selection - or restriction - a 'horizontal' subset retaining tuples which satisfy selection criteria

Binary operators supported are

> difference
> union
> intersection
> product
> join - two forms
> quotient

The first four binary operators are like the corresponding set operations. Intersection, join and quotient are derived, and the other five are all primitive. Each of the eight is discussed separately below. Section 4.11 includes a discussion of their efficiency.

4.1. Projection

```
PROJ alg_primary OVER att_id_list
```

where

```
att_id_list ::= att_id [, att_id_list]

att_id ::= [ qualifier . ] att_name

qualifier ::= rel_name |
              alias
```

For example, if a list of cities in which suppliers are located is required, any of

```
PROJ SUPPLIER OVER CITY

PROJ SUPPLIER OVER SUPPLIER.CITY

PROJ SUPPLIER=S OVER CITY

PROJ SUPPLIER=S OVER S.CITY
```

could be used. In this instance the first form is sufficient. Note that once an alias has been declared, it effectively replaces the name of the parent relation. Thus

```
PROJ SUPPLIER=S OVER SUPPLIER.CITY
```

would be flagged as an error. During parsing a list of the attributes to be retained is generated. A new relation node is then created (in the transient list) with appropriate values and header. At the same time the attribute list is updated to include offsets of each attribute in the source and destination tuples, and possibly compressed when consecutive attributes are encountered. Figure 4.2 shows the two forms of the list for the expres-

sion

```
PROJ SUPPLIER OVER SNAME, CITY, SNUM
```

The structure of the nodes here (and for selection) is given in Figure 4.1. Projection is the only algebraic operation which does not generate its result in order. The operand relation is traversed using two functions, first and next, which maintain a pointer to the appropri-

```
struct pnode
{  int pn[5];
   char *sptr,                  /* not used in algebra */
        *dptr,                  /* to offset in result */
        *argp;              /* to literal in condition */
   relptr result;         /* only used in SQL grouping */
   struct pnode *lson,    /* both used for binary tree */
                *rson;        /* used for simple list */
} pnode, *pnodeptr;
```

Figure 4.1 - Expression Node Structure

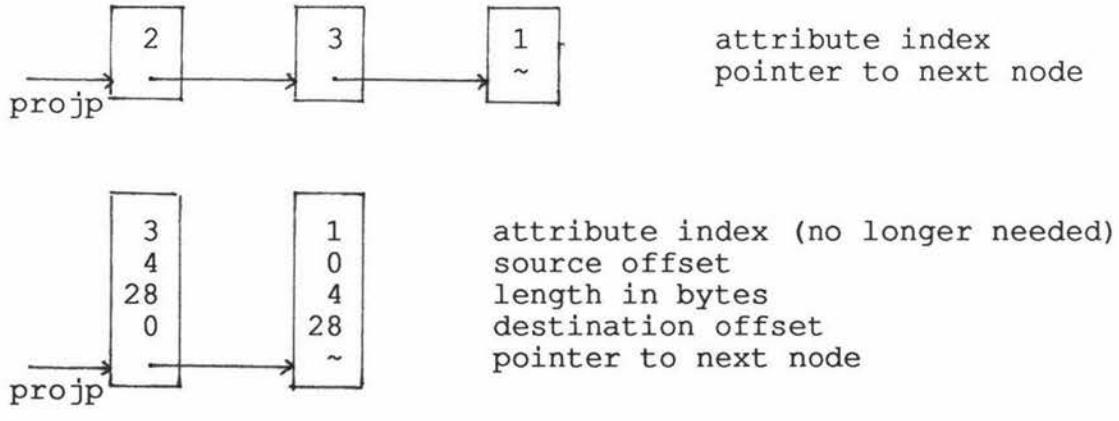


Figure 4.2 - Projection List Before and After Compression (unused fields omitted)

ate source tuple, and return the boolean value true if the appropriate tuple is found, and false if not, i.e. respectively an empty relation or no more tuples in the next direction. For each pnode in the list, data from the source tuple is copied to the destination tuple (dptr having been previously set to the correct offset position). A lookup operation determines whether the resulting tuple already exists, and if not it is inserted in the correct position as described in Chapter 3. In an attempt to minimize the number of lookups executed, a copy of the last tuple added is held, and each new candidate tuple is compared with this copy. (This reflects the fact that some projections will generate a sequence of duplicate candidates, e.g. the expression

```
PROJ SPJ OVER SNUM, PNUM
```

whenever a supplier supplies the same part to more than one job. Admittedly this approach may be criticised, and any advantage is the exception rather than the rule.) In Algorithm 4.1, A and C are the operand and result relations, tA is (a pointer to) the current source tuple, tC is (a pointer to) the candidate tuple for the result, and tL is (a pointer to) the last tuple added. (Appendix A gives an outline of the C-like language used to express algorithms. To simplify the algorithms further the distinction between a pointer and the object addressed is not made explicit.) Recall that the third parameter to find_tuple indicates the action, here add tuple tC if it is not found.

```
more = first(A), tL = "";
while more
   tC = proj(tA, projp);
   if tC != tL
      tL = tC, find_tuple(C, tC, '+');
   more = next(A);
```

Algorithm 4.1 - Projection

Despite the reservations expressed above, the algorithm performs well for the volumes of data used in the student environment, and the improvement possible with 'real' data through maintaining indexes is conjectural, unless there is already an index on the (list of) attribute(s) required. In this case only the index need be accessed. This is one place where the distinction between permanent relations (which have a primary key) and temporary relations (which do not) is counter-productive. If a relation has a primary key, the lookup need only consider the number of attributes in that key, not all attributes, as here.

In further discussion in this document the notation R[a] (where a is an attribute list) will sometimes be used for

PROJ R OVER a

4.2. Selection

SEL alg_primary WHERE sel_expr

where

```
sel_expr ::= sel_term [ '|' sel_expr ]

sel_term ::= sel_factor [ & sel_term ]

sel_factor ::= att_id relop att_id |
               att_id relop fn ( att_id ) |
               att_id relop literal |
               ( sel_expr )

fn ::= AVG |
       MAX |
       MIN |
       SUM
```

and literal is of the appropriate data type (with character data enclosed in either single or double quotes). For example, the expression

```
SEL SUPPLIER WHERE CITY = "LONDON"
```

finds suppliers located in London.

During parsing a binary tree is constructed (using pnodes) in which the leaves represent simple relation conditions and higher nodes contain representations of logical and/or; the condition above results in the single node of Figure 4.3. The operand is scanned in first, next order as described for projection. On each iteration the candidate tuple pointer and the root of the expression tree are passed to a recursive boolean function; if the result is true the tuple is added to the result relation after any existing tuples. (Since the operand was in sorted order and contained no duplicates there is no need for a lookup.) This is expressed in Algorithm 4.2.

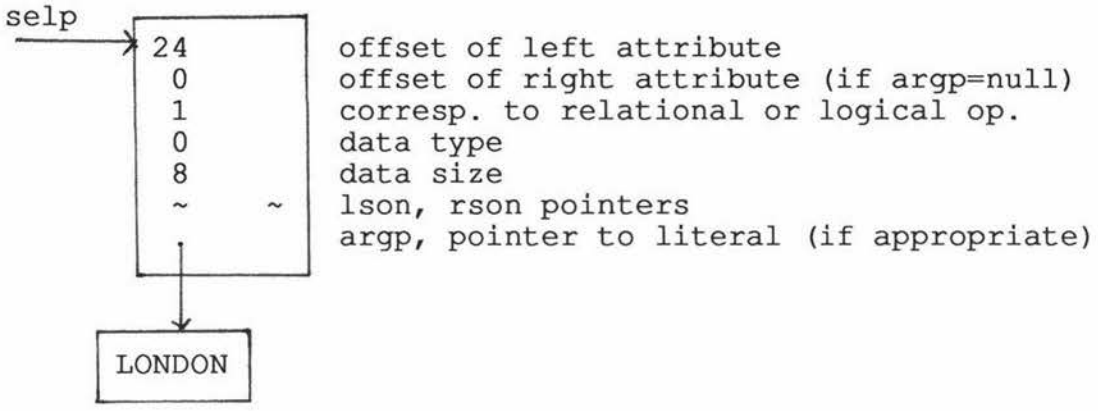


Figure 4.3 - Data Structure for Simple Selection

```
more = first(A);
while morea
   if satisfy(selp)
      add(C, tA);
   more = next(A);
```

Algorithm 4.2 - Selection

In further discussion in this document the notation R[c] (where c is a selection expression) will sometimes be used as an abbreviation for

```
SEL R WHERE c
```

4.3. Union

```
UNION alg_primary WITH alg_primary
```

The standard set-theoretic definition of set union is

$$A \cup B = \{x: x \in A \mid x \in B\}$$

which becomes for relational algebra 'tuples which are in relation A or in relation B (or possibly in both)'.

For the union operator to make sense the two operand relations must be 'union-compatible', i.e. have the same number of attributes which must be pairwise based on the same domain.

Since relations are stored in sorted order, a single pass through both operands is required, and the result is also generated in order. Using the first and next functions already introduced results in the algorithm given in Algorithm 4.3.

4.4. Difference

DIFF alg_primary WITH alg_primary

```
morea = first(A), moreb = first(B);
while morea | moreb
   if morea
      if moreb
         if tA < tB
            add(C, tA), morea = next(A);
         else if tA = tB
            add(C, tA), morea = next(A), moreb = next(B);
         else
            add(C, tB), moreb = next(B);
      else
         add(C, tA), morea = next(A);
  else
     add(C, tB), moreb = next(B);
```

Algorithm 4.3 - Union

The standard set-theoretic definition of set difference is

$$A - B = \{x: x \in A \ \& \ x \notin B\}$$

which becomes for relational algebra 'tuples which are in relation A and which are not in relation B'. Again, for the operation to make sense, the operands must be union-compatible, as described in Section 4.3.

As for union, a single pass through both relations is sufficient. (See Algorithm 4.4.)

4.5. Intersection

INTER alg_primary WITH alg_primary

The standard set-theoretic definition of set intersection is

$$A \cap B = \{x: x \in A \ \& \ x \in B\}$$

which becomes for relational algebra 'tuples which are in relation A and are also in relation B'. Intersection is a

---

```
morea = first(A), moreb = first(B);
while morea
   if tA < tB | !moreb
      add(C, tA), morea = next(A);
   else if tA = tB
      morea = next(A), moreb = next(B);
   else
      moreb = next(B);
```

Algorithm 4.4 - Difference

---

derived operation, since

$$A \cap B = A - (A - B)$$

i.e. the removal from relation A of tuples which are not in relation B leaves tuples which are. Again a simple single-pass algorithm is given (see Algorithm 4.5.)

4.6. Product

TIMES alg_primary BY alg_primary

The standard set-theoretic definition of a Cartesian product is

$$A \times B = \{(x,y) : x \in A \ \& \ y \in B\}$$

which forms ordered pairs by combining each element of set A with each element of set B. Elements of sets may themselves be compound. For relational algebra a new relation is formed by concatenating each tuple from relation A with each tuple from relation B, so that

---

```
morea = first(A), moreb = first(B);
while morea & moreb
   if tA < tB
      morea = next(A);
   else if tA = tB
      add(C, tA), morea = next(A), moreb = next(B);
   else
      moreb = next(B);
```

Algorithm 4.5 - Intersection

$$a(A \times B) = a(A) + a(B)$$

$$s(A \times B) = s(A) + s(B)$$

$$n(A \times B) = n(A).n(B)$$

where a(A), s(A) and n(A) are respectively the arity, the size of a tuple and the number of tuples in relation A. There is no corresponding simple formula for the storage requirement of a product; however if p(A) is the number of pages needed to hold relation A, we can say that p(A X B) has lower bound the smaller of p(A) and p(B), and upper bound approximately n(A).p(B) + n(B).p(A).

The header for the product is formed by concatenating the two separate headers, and increasing the aoffset field of each attribute from the second factor by the size of the first factor (less the extra reference-count integer).

Again, using the sorted order of the factor relations, MURDER gives a single-pass algorithm (see Algorithm 4.6.)

```
morea = first(A);
while morea
   moreb = first(B);
   while moreb
      add(C, concat(tA, tB)), moreb = next(B);
   morea = next(A);
```

Algorithm 4.6 - Product

4.7. Join

Of the many possible variants of the join operation, MURDER supports two:

- an extension of the theta-join
- the natural join

4.7.1. Theta Join

The theta-join is a derived operation which concatenates tuples from two relations for which nominated attributes are in a specified relationship. This can be seen as

```
(A X B)[a θ b]
```

where a and b are compatible attributes of relations A and B respectively, and θ is a relational operator, e.g. <, = etc. As an example, the attempt to find pairs of suppliers and jobs located in the same city using

```
SEL [TIMES SUPPLIER BY JOB] WHERE CITY = CITY
```

fails - the result of the nested product is a relation in which the third and sixth attributes are both named CITY, but in parsing the selection expression the first CITY attribute would be found twice, and in fact the selection achieves nothing. This is the reason for introducing attribute qualification, and what is required is

```
SEL [TIMES SUPPLIER BY JOB]
WHERE SUPPLIER.CITY = JOB.CITY
```

The earlier version allowed attribute indexing as a

simpler alternative:

SEL [TIMES SUPPLIER BY JOB] WHERE !3 = !6

Here the notation '!3' indicates the third attribute. Attribute indexing is a simple way of coping with this situation where more than one attribute has the same name, a situation not normally permitted but which may occur as the result of another operation, as here. It is the less satisfactory of the two approaches - although it is simpler to implement - and has inherent dangers. Suppose that before this expression is evaluated someone had replaced the SUPPLIER relation with

SUPPLIER(SNUM, SNAME, STATE, CITY)

The request using indexing would now fail because the STATE and JNAME attributes are based on different domains. For this reason, indexing, and indeed any form of expression which presupposes anything about the condition of the database - as distinct from facts such that both the SUPPLIER and JOB relations have an attribute named CITY - should be avoided.

MURDER introduces the form

JOIN alg_primary WITH alg_primary WHERE join_expr

as a convenient shorthand for this selection of a product; join_expr is defined like sel_expr, except that literals are not appropriate. Our example query can now be expressed

```
JOIN SUPPLIER WITH JOB
WHERE SUPPLIER.CITY = JOB.CITY
```

(The Prime version allows

```
JOIN SUPPLIER WITH JOB WHERE CITY = CITY
```

in which the apparent ambiguity in the join expression is resolved by having attributes on the left of a relational operator implicitly qualified by the first operand, and those on the right by the second.)

Note the MURDER extension which allows full expression syntax instead of only a single term.

The implementation builds a binary tree for the join expression as described for selection, and passes the root of this tree to the procedure used for product. The inner loop in the Algorithm 4.6 is changed to

```
while moreb
   if selp = null | satisfy(selp)
      add(C, concat(tA, tB));
   moreb = next(B);
```

This approach means that it is not necessary to build the product as an intermediate step, and is taken because of the potentially very high cost of the product operation. For this form of join, as for product, no lookups are required.

4.7.2. Natural Join

When the join expression consists of a single condition with relational operator '=', as in the example just

described, one column in the result will be duplicated, and in this case the natural join might be more appropriate. The natural join eliminates this duplication. It can be seen as

```
(A X B)[a = b]['everything except b']
```

which projects out the duplicated column. MURDER uses the form

```
JOIN alg_primary WITH alg_primary OVER att_name [, att_name]
```

If two att_names are given the first is applied to the first operand and the second to the second operand; if only one is given it is applied to both operands. The attributes must be compatible. Thus

```
JOIN SUPPLIER WITH JOB OVER CITY
```

would give the same result as

```
PROJ [JOIN SUPPLIER WITH JOB
      WHERE SUPPLIER.CITY = JOB.CITY]
OVER SNUM, SNAME, CITY, JNUM, JNAME
```

This problem of duplicate attribute names might have been avoided by naming the city attributes SCITY and JCITY (although this to some extent obscures the information that both are drawn from a common domain). But since it is sometimes necessary to join a relation with itself the underlying problem still remains; qualification by the relation name is no help if both are the same, hence an aliasing mechanism is required. One way to find suppliers of more than one part is

```
PROJ [JOIN SP WITH SP=SPX
      WHERE SP.SNUM = SPX.SNUM & SP.PNUM > SPX.PNUM]
OVER SP.SNUM
```

(It is recommended that an alias reflect the name of the relation affected.)

Note that even though projection is involved, lookup calls are never required. The inner loop of Algorithm 4.6 is further modified to give Algorithm 4.7.

In a database which supports the foreign key concept, it might reasonably be expected that joining a referencing relation to a relation it references over the foreign key would be a common processing requirement; indeed this could be seen as the natural join, since the purpose of a foreign key is to guarantee that such a join can be executed without loss of data from the referencing relation (which would occur if no target tuple exists, so that the

---

```
morea = first(A);
while morea
   moreb = first(B);
   while moreb
      if selp = null | satisfy(selp)
         tC = concat(tA, tB);
         if projp = null
            add(C, tC);
         else
            add(C, proj(tC, projp));
      moreb = next(B);
   morea = next(A);
```

Algorithm 4.7 - Product Modified to Support Joins

---

current tuple does not participate in the join). It is surprising to find no specific use of foreign keys made by algebraic operators, and it is a pity that the term 'natural join' is not used for this, e.g.

```
JOIN rel_name WITH rel_name
   OVER KEY [ att_name_list ]
```

in which the qualified form would only be necessary when, as sometimes occurs, a relation has more than one reference to the same relation. (But see Section 8.2.)

4.7.3. Other Joins

Several other types of join have been developed. One, the semi-join, only retains tuples from the first operand which would appear as 'prefixes' in the result of an equi-join with the second operand. This form is currently not supported directly, although it is used in the optimising of both calculus and SQL. The result has the same attributes as the first operand, and now the inner loop of Algorithm 4.7 is terminated either by exhausting the second operand, or by finding in the second a tuple which would cause the current 'prefix' tuple to be included in the result:

```
done = false, moreb = first(B);
while moreb & !done
   if satisfy(selp)
      add(C, tC), done = true;
   moreb = next(B);
```

All these joins lose information, in the sense that some tuples may not appear in the result. The so-called 'outer'

joins address this problem by extending such tuples in the appropriate direction by null values. Thus there are three forms depending on whether all information from the first or second operands or both is to be retained in the result.

4.8. Quotient

DIV rel_name BY rel_name

This operation is so called because it is in a sense the inverse of product:

$$A \times B \div B = A$$

although it is generally not true that

$$A \div B \times B = A$$

Informally, suppose $A = \{(x,y)\}$ and $B = \{y\}$; then

$$A \div B = \{x: \forall y \in B((x,y) \in A)\}$$

i.e. the quotient contains all 'prefixes' from A which occur in A associated with at least all the 'suffixes' from B. Formally, suppose that $A = \{(X,Y)\}$ and $B = \{Y\}$ where X and Y now represent attribute lists. Define the image set of an X-value x over A as

$$g(x,A) = \{y: (x,y) \in A\}$$

Then

$$A \div B = \{x: x \in X \ \& \ g(x,A) \supseteq B\}$$

Note that

$$a(A \,/\, B) = a(A) - a(B)$$

$$s(A \,/\, B) = s(A) - s(B)$$

$$n(A \,/\, B) \leq n(A) \,/\, n(B)$$

(As for the product there is no corresponding simple formula for the storage requirement of the quotient.)

An example should make the process clearer; suppose that

$$A = \{(a,b,1),(a,b,2),(a,b,3),(c,d,1),(e,f,1),(e,f,3)\}$$

$$B = \{1,3\}$$

so that $a(A) = 3$ and $a(B) = 1$, which gives $a(A \,/\, B) = 2$.

$$g((a,b),A) = \{1,2,3\} \supseteq B$$
$$g((c,d),A) = \{1\} \not\supseteq B$$
$$g((e,f),A) = \{1,2\} \supseteq B$$

$$A \,/\, B = \{(a,b),(e,f)\}$$

This example could be classified as a '3 by 1' division. MURDER supports the general m by n divide operation, and requires that

$$m > n$$

the last n attributes of the dividend be pairwise compatible with the attributes of the divisor

(It was a deficiency of Massey's original RDBMS that division was restricted to the 'n by 1' case. Notice that division by an empty relation gives a result containing all prefixes from the dividend, and that the quotient is empty if the dividend is empty, although an empty quotient may also result from other conditions.)

Division is a derived operation; an alternative definition in terms of primitive operators is

$$A / B = A[b'] - (A[b'] \times B - A)[b']$$

where [b'] denotes projection over the attributes not contained in B (for the MURDER implementation the first m - n attributes). Working from the inside out finds in turn

all prefixes	(projection)
all possible prefix-suffix combinations	(product)
possible combinations not in A	(difference)
prefixes from these missing combinations	(projection)
prefixes in the quotient	(difference)

The cost of executing this algorithm is the sum of the costs of two projections, two differences and one product. MURDER, taking advantage of the sorted physical order in which tuples are stored, implements a direct version of division, shown in Algorithm 4.8. (As before tA and tB refer to the current tuples in relations A and B respectively; pA and sA refer to the prefix and suffix of tA, and tC is a candidate tuple for the quotient.) This algorithm costs less than the equivalent product. (A further advantage, discussed in Chapter 5, is that it is readily modified to the case where

$$n(C) < n(A) - n(B)$$

which arises frequently during the evaluation of a calculus expression - see Section 5.4.1.)

```
morea = first(A);
while morea
   tC = pA, moreb = first(B);
   if moreb
      ok = TRUE;
      while ok & moreb
         if sA < tB
            morea = next(A),
            ok = morea | tC = pA
         else if sA = tB
            morea = next(A), moreb = next(B);
            if moreb
               ok = morea & tC = pA
         else
               ok = FALSE;
      if ok
         add(C, tC);
   else
      add(C, tC);
   while morea & tC = pA
      morea = next(A);
```

Algorithm 4.8 - Quotient

4.9. Updates

The operators described so far have been concerned with identifying data which satisfies given criteria. It must be emphasized that data retrieval is only one of several possible uses for algebraic expressions. The same expression can be used in a variety of contexts, e.g. to define the scope of an update or delete operation, to specify access rights for a user, or to define integrity constraints. See also the discussion of the MODIFY statement in Section 3.3.2.

4.10. Discussion

Since the result of each of the eight algebraic operators described above is a relation (formally the set of relations is closed for these operators) the algebra is inherently recursive, i.e. wherever a relation appears in a relational expression it may be replaced meaningfully by another such expression. These derived relations inherit attribute names and domains from the operands from which they are derived. Whenever this inheritance produces the duplication of an attribute name, provision must be made for distinguishing between the different attributes. The aliasing described for MURDER is adequate, but sometimes produces surprising results. If we

```
LIST JOIN SP WITH SP=SPX OVER SNUM
```

the heading for the result looks like

```
SP          SP          SPX
SNUM        PNUM        PNUM
----------- ----------- -----------
```

because aliasing is implemented by replacing the parent name fields in the header of the aliased relation. This is the reason for the earlier comment that an alias should reflect the name of the affected relation. Incidentally, this corresponds very closely to the use of tuple variables described for calculus in Section 5.4.1.

As with any form of expression, the user has to learn the 'tricks of the trade', i.e. techniques which experience has shown are likely to be helpful in given situations.

Often a query is formulated using a difference in which the second operand is the negation of the original query - A previous example showed how to find suppliers of more than one part; finding suppliers of only one part becomes

```
DIFF [PROJ SUPPLIER OVER SNUM]
WITH [the previous query]
```

This applies even to apparently simple queries:

```
PROJ [SEL SPJ WHERE PNUM="P1"] OVER SNUM
```

gives suppliers of part P1; but

```
PROJ [SEL SPJ WHERE PNUM<>"P1"] OVER SNUM
```

doesn't give, as the novice might expect, suppliers who don't supply P1 - it gives suppliers of at least one part other than P1, and what is required is

```
DIFF [PROJ SUPPLIER OVER SNUM]
WITH [PROJ [SEL SPJ WHERE PNUM="P1"] OVER SNUM]
```

Again, queries formulated using the word 'all' often require the quotient operator. Suppliers who supply the same part to all jobs are given by

```
DIV SPJ
BY  [PROJ J OVER JNUM]
```

although the solution to the apparently simpler query merely to find who supplies all jobs is

```
DIV [PROJ SPJ OVER SNUM, JNUM]
BY  [PROJ J OVER JNUM]
```

Thus formulating an algebraic expression requires detailed

knowledge of both the structure of the database and the semantics of the various operators. As queries become more complex, the initial response of many students that "algebra is easy", probably encouraged by its procedural nature, undergoes major revision. A solution to the quite reasonable query "What jobs could be supplied entirely by supplier S2?" is

```
PROJ [JOIN [DIFF [PROJ J OVER JNUM]
            WITH [PROJ [JOIN SPJ
                        WITH [DIFF [PROJ P OVER PNUM]
                              WITH [PROJ [SEL SP
                                          WHERE SNUM="S2"]
                                    OVER PNUM]]
                        OVER PNUM]
                  OVER JNUM]]
      WITH J
      OVER JNUM]
OVER JNAME
```

which is anything but obvious! Working from the inside out gives successively

```
SP-tuples for supplier S2
PNUM values from supplier S2
PNUM values which S2 doesn't supply
SPJ-tuples for parts S2 doesn't supply
JNUM values for jobs using parts S2 doesn't supply
JNUM values using only parts S2 does supply
JNAME values as required
```

4.11. Efficiency

Although it is not a factor in a memory-resident system, the discussion of efficiency is in terms of page references. However, as the more common approach is to use auxiliary storage, these comments are still relevant to current practice.

For projection, each new tuple requires a lookup to determine whether it already exists, and if not where it should be placed. An index on the primary key would facilitate this, and in MURDER the relation itself acts as such an index, since here (and in the result of all operations) the primary key is not known. There is some evidence (Bitton[83]) that it is worth while constructing such an index if it does not exist, and if the projection is a common processing requirement maintaining it would certainly pay. It should, however, be noted that there is an additional problem inherent in using such indexes: either the location of a tuple is fixed in the physical database (this fixing of position is sometimes referred to as 'pinning'), or an indirection mechanism must be used, such as that described for System R in Ullman[82].

In the absence of an index on (even part of) the selection condition, the selection algorithm is optimal - each tuple is accessed once. Similarly the join operations might benefit from the use of indexes.

The set operations union, difference and intersection, are all expressed as single-pass algorithms. For union this is nearly optimal, even with indexes - each page of each operand is accessed once. If indexes on the primary key of the two operands are available, only pages from the second operand which contribute tuples to the result need be accessed, but any saving here is offset by accesses to index pages. For the other two there is scope for more savings, as tuples from the second operand never appear in

the result of either intersection or difference.

For product there is nothing to be gained by using indexes. The number of page references required by Algorithm 4.6 is p(A).p(B), not counting writing pages of the result. However this algorithm generates its result in order, so that there is no need for either lookup or page-splitting operations. If the smaller factor is read into memory, the number of page references becomes p(A)+p(B), corresponding to MURDER in the case when one factor fits into a single page. Similar comments apply to division. If the divisor fits into a single page, the number of page references is p(A)+1, but it may be as high as n(A).p(B). (In this case the more direct algorithm would be better.) A primary key index of the divisor should be smaller than the divisor itself, and so would improve the execution of Algorithm 4.8, giving up to n(A).p(bI), the number of pages in B's index.

The original design decision to store tuples in sorted order has also been shown to be helpful, leading as it does to simple and efficient (in some cases optimal) algorithms for the various operators. The next chapter shows that the same advantages accrue for the relational calculus. The use of indexes would however improve the performance of the more common operators where real data volumes are used, but would also require more complex memory management techniques.

CHAPTER 5

MURDER Calculus

5.1. Relational calculus

An alternative to relational algebra for specifying manipulation of the database is provided by relational calculus. Codd[71] informally proposed a language (called Data Sub-Language ALPHA) based on predicate calculus. Both relational algebra and (tuple) relational calculus were formally introduced in Codd[72]. This paper also demonstrated the equivalence of the two approaches by giving an algorithm for converting a calculus expression to the equivalent algebra. Alpha was never implemented, although QUEL (used with the Ingres Relatonal DBMS) is similar. (The calculus described here, and implemented in MURDER, is closely related to that defined in Codd[72].) Although the expressions look different from the algebra they are equivalent, the apparent differences being mainly in the mode of expression; whereas algebra is prescriptive, in that an algebraic expression defines how to obtain the data you want, calculus is descriptive, with the user specifying the characteristics of the required data, and leaving it to the system to decide how it is to be obtained. Another common way of distinguishing between the two approaches is to say that algebra is procedural - in the same way as common programming languages, where the procedure for evaluating an algorithm is explicitly coded - and calculus is non-procedural.

As a simple example of this, consider again the query "What suppliers supply part P1?", for which the algebraic expression is

```
PROJ [SEL SP WHERE PNUM="P1"] OVER SNUM
```

and the equivalent calculus expression is

```
(x{SP.SNUM): x.PNUM="P1"
```

A calculus expression is of the form

```
( target_list ) [ : qualification ]
```

Although the notation is mathematical it is easily decoded. (Note that this should be

```
{ ( target_list ) [ : qualification ] }
```

showing explicitly that the result is a set, but the braces are usually omitted. The term 'qualification' is used for the more formal 'predicate'. The meaning of the query above is given in Section 5.1.2.) The components of this form are discussed in the next paragraphs, before the implementation is described.

5.1.1. The target list

The target list specifies the attribute(s) required in the result of the expression. In the example above, x is known as a 'range' or 'tuple' variable (hence the term tuple calculus, as distinct from domain calculus in which variables represent domains - a better term would be attribute calculus since attributes are specific to relations while

domains are not), and SPJ is its range; each range variable must be associated with a range the first time it is used. Just as in Pascal

```
var x : integer;
```

specifies that integer variable x may hold any value from an implementation dependent set of integers, so

```
x{SP
```

specifies that tuple variable x ranges over relation SP ('{' is a keyboard approximation to the set membership notation 'Є'), i.e. the current value of relation SP in the database defines both the domains from which the atomic values in x will be drawn, and also the set of possible combinations of these values. If all attributes of a range are required this is all that is needed, otherwise the qualified form is used. When more than one attribute is to be retained from the same range, the range is only given once, on the first use of the range variable. To retain both the SNUM and JNUM attributes from SPJ in a query, the target list could be

```
(x{SPJ.SNUM, x.JNUM)
```

In fact this is the complete expression for the calculus equivalent of

```
PROJ SPJ OVER SNUM, JNUM
```

Note that since the calculus always uses variable names to qualify attribute references, the problem of ambiguity

discussed for algebra is removed.

This is the first difference between MURDER calculus and that outlined by Codd: the range of all variables is introduced on their first appearance in the expression; Codd introduced the ranges of 'free' variables (those in the target list) using monadic predicates at the beginning of the qualification. For this query the expression would be

```
(x.SNUM, x.JNUM): x{SPJ
```

The difference appears largely notational but the implications are wider. In the MURDER calculus each expression is entirely self-contained, but Alpha used a separate 'range' statement to define the range of a variable, removing these monadic predicates (called range terms) from the qualification:

```
RANGE SPJ x
GET W (x.SNUM, x.JNUM)
```

where W names the work-space to hold the result. Note that now the range of the variable is semi-permanently defined outside of the expression in which it is used.

More than one variable may appear in the target list; the calculus equivalent of the product of SUPPLIER with JOB is

```
(s{SUPPLIER, j{JOB)
```

5.1.2. Simple Qualification

The first example in this chapter uses simple qualification to indicate which tuples participate in the result. Thus this example

```
(x{SP.SNUM): x.PNUM="P1"
```

may be read as "Retain SNUM values from SP-tuples x such that x's PNUM value is P1". Of course duplicates are eliminated.

Although calculus equivalents of all algebraic operators are not given, consider the theta-join used in Section 4.7.1:

```
JOIN SUPPLIER WITH JOB
WHERE SUPPLIER.CITY = JOB.CITY
```

The following calculus equivalent shows the explicit qualification:

```
(s{SUPPLIER, j{JOB): s.CITY = j.CITY
```

5.1.3. Range Expressions

Before going on to more complex qualification expressions, here is the syntax of range expressions:

```
range_expr ::= range_term [ '|' range_expr ]

range_term ::= range_factor [ & [ ^ ] range_term ]

range_factor ::= relation_name |
                 ( range_expr ) |
                 '[' calc_expr ']'
```

in which the three operators |, &^ and & correspond to set union, difference and intersection respectively. As with algebra, recursion is indicated by brackets. (Recall that '[' denotes an instance of the left-bracket character; the quotes distinguish this from the indication of an optional part of a syntactic form.)

DIFF A WITH B

is represented by

(x{A &^ B)

(i.e. x is a tuple of A and not a tuple of B; of course relations A and B must be union-compatible).

5.1.4. Further Qualification

Returning to the query about who supplies part P1, suppose the supplier name is required rather than the number. From what was said in Section 5.1 the user might expect something like

(s{SUPPLIER.SNAME, x{SP): s.SNUM=x.SNUM & x.PNUM="P1"

but this retains all attributes of SP in the result. Instead the solution is

(s{SUPPLIER.SNAME):]x{SP(s.SNUM=x.SNUM & x.PNUM="P1")

where the right-bracket is a keyboard approximation for the existential quantifier '∃', which is read 'there exists'. This query then reads "retain the SNAME value from SUPPLIER-tuples s such that there exists an SP-tuple

x with x's SNUM value equal to y's SNUM value and y's PNUM value equal to P1".

(In Alpha this might be expressed simply

```
GET W (SUPPLIER.SNAME):
        ]SP(SUPPLIER.SNUM=SP.SNUM & SP.PNUM="P1")
```

or `GET W (s.SNAME): ]x(s.SNUM=x.SNUM & x.PNUM="P1")`

provided in the latter case that at some earlier point the statements

```
RANGE SUPPLIER s
RANGE SP x
```

had been executed. This dynamic declaration of variable ranges, which persist until the variable is redefined, gives an apparent simplification of the calculus expression, but only at the expense of removing from the expression information critical to its meaning. Note that in the former case SUPPLIER and SP are both required to fulfil two distinct functions, viz. to specify the range and as a range variable. These functions are separated in MURDER.)

Names of suppliers who don't supply this part can be obtained by simply negating the qualification:

```
(x{SUPPLIER.SNAME): ^]y{SP(x.SNUM=y.SNUM & y.PNUM="P1")
```

reading '^]' as 'there does not exist'.

The universal quantifier '∀' is also available. It is approximated '@' (read 'for all') and may be negated. To find suppliers of only one part:

```
(x{SP.SNUM): @y{SP (x.SNUM=y.SNUM => x.PNUM=y.PNUM)
```

The symbol '=>' denotes logical implication. This query reads "retain SNUM values from SP-tuples x such that for all SP-tuples y if the SNUM values are the same then the PNUM values are the same". According to the normal rules of Boolean algebra

```
  p => q = ^p | q
^(p & q) = ^p | ^q
^(p | q) = ^p & ^q
```

Negated quantifiers are resolved using the equivalences

```
^∃x(p) = ∀x(^p)
^∀x(p) = ∃x(^p)
```

(A variable is passed into the expression-parsing procedure indicating that the components of the expression tree are to be negated.) This qualification may be expressed in a number of ways:

```
    @y{SP (x.SNUM<>y.SNUM | x.PNUM=y.PNUM)
or ^]y{SP (x.SNUM=y.SNUM & x.PNUM<>y.PNUM)
```

the last of which is read "... such that there does not exist an SP-tuple y with the same SNUM value and a different PNUM value".

To find the names of these suppliers, the SUPPLIER relation must be introduced in the target list, with the two SP-based variables in the qualification, which now contains an extra 'join' term:

```
(s{SUPPLIER.SNAME): ]x{SP (s.SNUM=x.SNUM &
                    @y{SP (x.SNUM=y.SNUM => x.PNUM=y.PNUM))
```

It would be misleading not to point out here that negating a calculus expression is not always simply a matter of negating the qualification. Where

```
(x{SP.SNUM): x.PNUM="P1"
```

gives the numbers of suppliers who supply part P1,

```
(x{SP.SNUM): x.PNUM<>"P1"
```

does not give numbers of suppliers who don't; instead it gives numbers of suppliers of at least one part other than P1, and the correct expression is

```
(x{SP.SNUM): ^]y{SP(x.SNUM=y.SNUM & y.PNUM="P1")
```

which suggests that in some sense a better solution to the former query is

```
(x{SP.SNUM): ]y{SP(x.SNUM=y.SNUM & y.PNUM="P1")
```

since - as above - this can be negated. (This comment applies whenever such simple qualification is not based on the primary-key or an attribute functionally dependant on it, i.e. an alternate key. In SQL the scope for this anomaly is increased.)

As will already be apparent, relational calculus is not without its own techniques for formulating the answer to a query, and although the approach is different from that for algebra, the final expression may still be built up

step by step. (The last example extends the requirement from supplier numbers to supplier names in such a step; note that it is well-defined.)

Where for algebra a query involving 'all' is usually expressed using the quotient operator, now in calculus it will usually be expressed using the universal quantifier. For example to find who supplies all jobs:

```
(x{SPJ.SNUM): @j{JOB ]y{SPJ (x.SNUM=y.SNUM & j.JNUM=y.JNUM)
```

or who supplies the same part to all jobs by

```
(x{SPJ.SNUM): @j{JOB ]y{SPJ(x.SNUM=y.SNUM & x.PNUM=y.PNUM &
                            j.JNUM=y.JNUM)
```

The universal quantifier in turn often gives a 'natural' expression of a query using implication. The last example in Chapter 4 concerned jobs which could be supplied entirely by supplier S2; this requires that for a job to be included, for all SPJ-tuples with that job number it must be possible to find an SP-tuple with the same part number and in which the SNUM value is S2.

```
    (j{J.JNAME): @x{SPJ (j.JNUM=x.JNUM =>
                         ]y{SP (x.PNUM=y.PNUM & y.SNUM="S2"))
```

(Compare the algebra solution in Section 4.10!)

5.2. Codd's Algorithm

In Codd[72] the notion of 'relational completeness' of a data language is introduced to mean a language at least as expressive as relational algebra, and the completeness of

relational calculus is then demonstrated by giving an algorithm for converting a calculus expression to the eqivalent algebra. The core of the algorithm depends on the observation that any information derivable from a set of relations can be obtained from the product of those relations by some combination of selections and projections.

The steps in the conversion algorithm are summarised in Algorithm 5.1. Step 4 requires some justification, and the following is only an intuitive outline of the approach.

---

(1) Form a selection expression from the simple qualification expression (recall that in prenex-normal form all quantifiers come first leaving a simple expression last)

(2) Form the product of all ranges (in order)

(3) Apply the selection expression from (1) to the product from (2)

(4) Resolve any range-coupled quantifiers from right to left in turn:

```
if the quantifier is @
     divide the result of the last operation
        by the associated range
else
     project the result of the last operation
        over all attributes of ranges
           left of the associated range.
```

(5) Project the result of the last operation over attributes from the target list.

Algorithm 5.1 - Codd's Algorithm

The quantifier '∃' introduces a boolean predicate concerning the external and coupled ranges, and from Step 3 the only tuples remaining in the universe satisfy this predicate; thus the existence of a tuple in the universe corresponds to the truth of the predicate, viz. the existence of a tuple from the quantified range matching the 'prefix' (or external range), so that this prefix should be retained.

For the universal quantifier '∀' the predicate must be true for all the tuples in the coupled range, which is the divisor in the operation in this step. Since the existence of a tuple in the universe implies the truth of the predicate for that tuple, and since the quotient operator retains only prefixes for which the combination with (at least) all tuples in the divisor appear, it can be seen that a division does indeed represent the intended meaning of the predicate.

As an example consider the application of this algorithm to the last query, which in prenex-normal form is

```
(j{JOB.JNAME): @x{SPJ ]y{SP (j.JNUM<>x.JNUM |
                             x.PNUM=y.PNUM & y.SNUM="S2")
```

For convenience the attributes of the factor ranges are listed together with the associated attribute indexes (indexes are used for brevity only):

```
JOB j  JNUM   JNAME  CITY
         1      2     3

SPJ x  SNUM   PNUM   JNUM
         4      5     6

SP  y  SNUM   PNUM
         7      8
```

Step 1: form selection expression

```
s = !1<>!6 | !5=!8 & !7="S2"
```

Step 2: form universe

```
U = JOB X SPJ X SP
```

Step 3: apply selection expression to universe

```
T3 = U[s]
```

Step 4: resolve quantifiers (right to left)

```
']'   T2 = T3[!1..!6]    (i.e. !1,!2,!3,!4,!5,!6)

'@'   T1 = T2 / SPJ
```

Step 5: final projection (and back-substitution)

```
T = T1[!2]
  = (T2 / SPJ)[!2]
  = (T3[!1..!6] / SPJ)[!2]
  = (U[s][!1..!6] / SPJ)[!2]
  = ((JOB X SPJ X SP)[s][!1..!6] / SPJ)[!2]
  = ((JOB X SPJ X SP)[!1<>!6 | !5=!8 & !7="S2"]
                [!1..!6] / SPJ)[!2]
```

or, in algebra,

```
PROJ [DIV [PROJ [SEL [TIMES JOB
                      BY [TIMES SPJ
                          BY SP]]
                  WHERE JOB.JNUM<>SPJ.JNUM
                      | SPJ.PNUM=SP.PNUM & SP.SNUM="S2"]
            OVER JOB.JNUM,JNAME,CITY,SP.SNUM,SP.PNUM,SPJ.JNUM]
      BY SPJ]
OVER JNAME
```

(Only the last attribute in the projection list <u>needs</u> qualification, to distinguish it from the first, although

the fuller form above is preferable.) The starting-point for the calculus implementation in MURDER is this algorithm. Before going into this, however, one more point made by Codd[72] is considered.

The notion of a membership predicate to make the calculus relationally complete is suggested. To obtain, using algebra, all the cities having either suppliers or jobs, would require

```
CITY := UNION [PROJ SUPPLIER OVER CITY]
        WITH  [PROJ JOB OVER CITY]
```

Now in calculus union is achieved in the range expression, and since the union operation requires compatible operands, the calculus as it stands would require the sequence

```
SC := (s{SUPPLIER.CITY)
JC := (j{JOB.CITY)
CITY := (x{SC | JC)
```

To be able to do this in a single expression (which is the object of full relational completeness) Codd's approach gave

```
CITY := (x{SUPPLIER[CITY] | JOB[CITY])
```

in which the relations in the range expression are replaced by the result of a projection of those relations, called a membership predicate. MURDER extends this by allowing the replacement to be by any calculus expression; the result

```
CITY := (x{[(s{SUPPLIER.CITY)] | [(j{JOB.CITY)])
```

is not elegant (largely due to the decision to use a mathematical notation which introduçes a calculus expression by '(' rather than a keyword such as GET or RETRIEVE), but the power is considerably extended, and (as is described below) the potential for improved performance is enhanced.

5.3. MURDER Implementation

As the expression is parsed, a doubly-linked list of nodes is constructed to maintain information about the various ranges and associated quantifiers. The structure of these calculus nodes is given in Figure 5.1.

Each time a range expression is introduced (by the character '{'), the list so far is checked to ensure uniqueness of variable names, then the range expression is evaluated

```
struct calculusnode
{ char varname[NAMESIZE];
  relptr relp;                  /* to associated range */
  int cumatt,                   /* # attributes so far */
      cumsize,              /* cumulative size so far */
      quant;                      /* 1=], 2=@, else 0 */
  struct calculusnode
     *prevcal, *nextcal;       /* doubly linked list */
  pnodeptr alist,              /* these are described */
           slist;               /* under optimization */
}
```

Figure 5.1 - Calculus Node

and the result attached to the relp field of a new calculusnode linked to the tail of the existing list. While parsing the target list, a list of pnodes is constructed as described for projection in Chapter 2. In processing the qualifying expression, a binary tree is built as described for selection. However, there are three additional features:

(1) Each calculusnode maintains (using the alist field) a list of attributes from the associated range which are encountered. This applies to both the target list and the qualifying expression, and is similar to a projection list except that a count of the number of times each attribute has been met (whether in the target list or qualification) is included.

(2) The binary tree representation of the qualifying expression is constructed in such a way that during evaluation attributes from ranges occurring early in the expression are, as far as is possible, examined before attributes from later ranges.

(3) If a comparison with a literal is found a global variable is set to indicate that at least one of the variable ranges might be reduced by pre-selection.

(These points are further discussed in the next section on optimization.)

Once the expression has been parsed without error, evaluation begins, generally along the lines indicated by Steps

2-5 of Algorithm 5.1 above, but modified by the optimization measures described in the next paragraph. For the example query regarding jobs which could be supplied entirely by supplier S2, the data structures are given in Figure 5.2.

5.4. Optimization

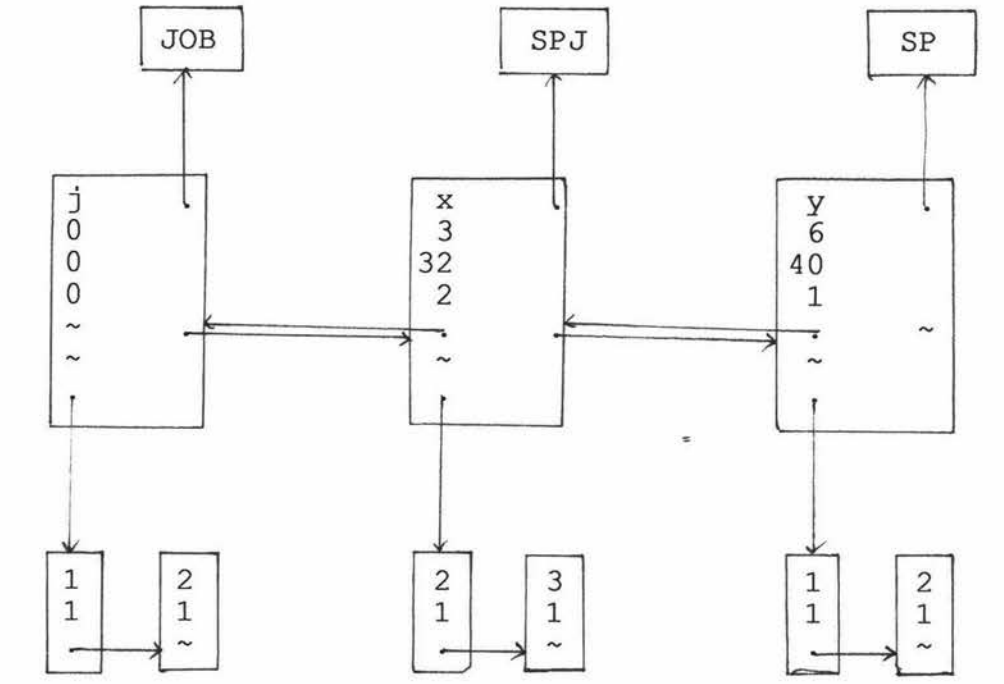
The raw implementation of Codd's reduction algorithm contains obvious inefficiencies, e.g. where two or more projections follow consecutively, only the last is required, and the cost of creating the universe by a sequence of binary products is exorbitant, especially when it is remembered that a significant proportion of tuples are likely to be eliminated immediately by applying the selection expression in Step 3.

The main areas of optimization addressed are

- eliminating unnecessary operations
- eliminating unnecessary computation
- reducing the size of intermediate relations

5.4.1. Eliminating Unnecessary Operations

This is attempted at several points. Instead of using the binary algebraic primitive for product, a tailored routine is provided, an iterative representation which generates a candidate tuple for the universe at each cycle. As for the algebraic join, rather than generate the whole product first and then select, the selection expression is applied



list of calculusnodes and associated structures

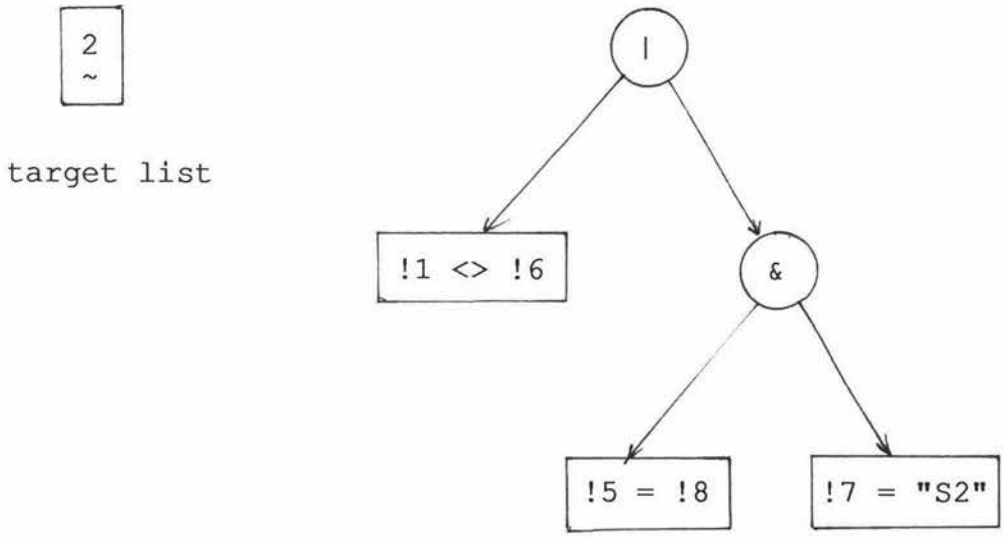


Figure 5.2 - Data Structures for Example Query

to this tuple (as described in the next paragraph). If the right-most quantifier(s) is(are) ']', the latter attributes in this tuple would immediately be projected out in the next cycle(s) through Step 4. These projections are eliminated by only saving the required 'prefix' of a tuple which satisfies the selection expression, and then skipping over any further tuple combinations which would produce the same prefix. Thus a single operation accomplishes Steps 2 and 3 and half of Step 4. (Using sample data this reduced the number of tuples in the universe from 4375 to 162, and the size of the universe from 65 pages to 2.)

The same approach is taken when resolving universal quantifiers in the rest of Step 4. If the sequence ']...]@' occurs, trailing attributes in the result of the division will similarly be projected out in subsequent cycles through this step. By only saving the prefix in this case, and by skipping over tuples in the dividend which would give the same prefix, these projections are also eliminated and processing time further reduced. This prefix is determined by scanning the calculusnode list forward until other than the representation of an existential quantifier is found, i.e. the next universal quantifier or the target list ranges.

Finally, if all attributes of ranges in the target list are to be retained, the final projection of Step 5 is omitted. Thus evaluation becomes simply

```
form the reduced universe

resolve any universal quantifiers

if necessary
   do the final projection
```

5.4.2. Eliminating Unnecessary Computation

The previous section has introduced this with the discussion of the modified division algorithm. What is left is eliminating unproductive applications of the selection expression to candidate tuples.

Recall that the tree representation of the selection expression was constructed so that 'early' comparisons could be evaluated before 'later' ones. When examining a tuple the maximum offset of attributes considered is stored. As soon as it is known that the tuple fails to satisfy the qualifying expression, the level of iteration is reduced until the offset of the first attribute in a factor is less than this maximum offset, since all further combinations using the same factor at this level will also fail at the same point. (In practice this reduces the number of tuples tested by an order of magnitude.)

5.4.3. Reducing Relation Size

Elimination of projections resulting from ']' quantifiers has already been discussed, so that here there remains only the problem of reducing the size of the factor relations in the universe as a way of further reducing both the size and the amount of computation needed to generate

the reduced universe.

When parsing the qualification expression a boolean variable was set to true if comparison with a literal was encountered. If this variable is true, the expression tree is examined before generating the universe. If such a comparison is found before the representation of '|' (logical or, but see discussion below), this node is pruned from the tree and applied instead to the slist pointer of the corresponding calculusnode. At the same time the attribute reference count in the corresponding alist node is decremented, and if it becomes 0 the node is removed from the list.

Now if any calculusnode has a non-null slist pointer, or if its attribute list does not reference all the attributes of the corresponding relation, the relation is replaced by the result of the indicated selection and/or projection. (Other data-structures are modified as required. In the discussion example this reduced the number of tuples in the universe before and after optimization to 1540 (4375) and 144 (162) respectively; numbers in parentheses refer to tuples before the first stage of optimization described in the previous section.)

Note that this is the equivalent of having the system do what the user might have done using recursion. The names of suppliers of part P1 are given by

```
(s{SUPPLIER.SNAME): ]x{SP(s.SNUM=x.SNUM & x.PNUM="P1")
```

For SUPPLIER, only SNUM and SNAME attributes are referenced (once each); for SP, attributes referenced, also once each, are SNUM and PNUM. But, since the simple expression is in conjunctive-normal form (factors joined by '&') the second factor, which only references the range of x, may be 'factored out', reducing the corresponding reference count to zero, and eliminating it from the corresponding attribute list. The ranges of s and x may thus be replaced by

```
s' = (s{SUPPLIER.SNUM, s.SNAME)
x' = (x{SP.SNUM): x.PNUM="P1"
```

respectively, giving

```
    (s{s'.SNAME): ]x{x'(s.SNUM=x.SNUM)
or  (s{[(s{SUPPLIER.SNUM, s.SNAME)].SNAME):
       ]x{[(x{SP.SNUM): x.PNUM="P1"]
          (s.SNUM=x.SNUM)
```

For simple queries and small data volumes this actually increases the cost; considerable analysis of test results would be needed to determine criteria by which it could be automatically decided whether an advantage is to be gained by this pre-projection/selection. Intuitively it is unlikely to be helpful if a relation is stored on a single page.

The condition mentioned above "before the representation of '|'" is weak, and would not detect a condition such as

```
(x.SNUM="S1" | x.SNUM="S2") & ...
```

A better approach is that hinted at above, viz.

> "whenever a factor in the qualifying expression in conjuctive-normal form contains only references to attributes of a single range"

or perhaps this is a case where the recursive approach could be used.

The final algorithm is given in Algorithm 5.2.

The steps in optimizing outlined are close to those suggested by Palermo[74], although this paper had not been seen at the time.

For the small sample data used for testing these steps did not produce a significant reduction in overall processing time, possibly because the cost of additional pre-processing offset any projected savings in the main processing. However the potential is clear: if only 20% of a relation is retained through pre-selection, up to 80% of both computation and space will be saved later; and if a projection reduces the size of a factor by 20% then there will be a cumulative reduction of up to 20% in the size of the universe.

5.5. Discussion

Although the calculus described here uses a formal mathematical notation, Codd argues that its natural mode of expression makes it an ideal target for a relational language. Alpha and the language presented in Date[86] present two approaches to a more English-like calculus,

```
                              /* determine prefix size = sA */
for (l=1, tcal=cnhead; tcal!=null; tcal=tcal->nextcal)
   offset[l++]=tcal->cumsize;
   if tcal->quant!=1
      sA=tcal->cumsize+rcal->relp->size;

                                        /* evaluate universe */
tcal=cnhead, l=1, more[1]=first(tcal->relp);
while (more[1])
   maxoffset=0;
                                     /* form candidate tuple */
   tA+offset[l]=t[l];
   while (tcal!=cntail)
      more[++l]=first((tcal=tcal->nextcal)->relp);
      tA+offset[l]=t[l];

   ok=(selp=null)|satisfy(selp);                  /* select */

   if ok
      add(A, tA);        /* and drop back to next prefix */
      while (offset[l]>=sA)
         l--, tcal=tcal->prevcal;
   else
               /* drop back to next possible combination */
      while (offset[l]>= maxoffset)
         l--, tcal=tcal->prevcal;

   more[l]=next(tcal->relp);
                              /* if no more at this level */
   while (l>1 & !more[l])                      /* drop back */
      more[--l]=next((tcal=tcal->prevcal)->relp);

                /* skip trailing existential quantifiers */
for (tcal=cntail; tcal->quant=1; tcal=tcal->prevcal);

                           /* resolve universal quantifiers */
for (; tcal->quant>0; )
   for (B=tcal->relp, sA=tcal->cumsize, tcal=tcal->prevcal;
        tcal->quant=1;
        sA=tcal->cumsize, tcal=tcal->prevcal);
   A=divide(A, B);

                                        /* final projection */
compress(projp);
if projp->rson!=null | resultsize < A->size
   A=project(A, projp);
```

Algorithm 5.2 - Calculus Universe

and the SQL language to be described in Chapters 6 and 7 is a third. Codd argues in favour of calculus on the basis of the following three points:

(1) calculus is more extendable than algebra to allow for arithmetic operations, built-in functions etc. It is easy to imagine a calculus expression in which the target list includes something like

    AMT = x.QTY * y.PRICE

    (which also introduces the idea of 'aliasing' attributes or expressions in the result); it is not so easy to imagine extending the syntax for projection to allow for this.

(2) calculus makes it easier to capture the user's intent (which is important for authorizing access, selecting indexes for optimizing, etc).

(3) calculus is closer to a natural language expression of a query.

Date questions these arguments on the grounds of the formal equivalence of the two approaches. It is our experience at Massey that students find algebra easier initially (although this may be because they meet algebra first), and those without a mathematical background retain this preference while admitting that devising an algebraic solution becomes less direct as queries become more involved. (We have not had sufficient experience with SQL to comment on it.)

Areas of difficulty with the calculus appear to be the concepts of free and bound variables, and quantifiers. (Variables in the target list are free, while those declared with range-coupled quantifiers are bound within the sub-expression the range predicate introduces.) The calculus syntax does however make the binding explicit; as is shown in Chapter 6 SQL sometimes obscures this (and indeed even claimed not to need these concepts), creating a situation in which the user may be caught out. Moving the quantifiers to the RANGE statements, as in Alpha, makes the binding less clear.

The performance of the optimised calculus compares favourably with the algebra. (Note that no attempt is made to optimize the algebra.) For the query analyzed in this chapter, algebra takes 24 page references using sample data, compared with 16 for calculus, or only 10 without pre-processing ranges. The advantage turns in favour of algebra (which tends towards producing smaller intermediate relations) as data volume increases.

CHAPTER 6

Introduction to SQL

Codd[71] introduced the calculus-based data sub-language Alpha, already mentioned briefly in Chapter 5. This language was never implemented, although QUEL (Query Language, used with the Ingres system developed at the University of California, Berkeley, and now released commercially by Relational Technology Inc.), is closely related. However, some of the more difficult formal concepts of a calculus, such as the calculus described in Chapter 5, are hidden in an attempt to make the language more accessible. A number of features of Alpha are therefore described as background, and occasionally the different approach adopted in Quel is mentioned.

In many ways SEQUEL (and its successors SEQUEL2 and SQL) represent a different approach to the same problem, viz. hiding these concepts - quantifiers and free and bound variables - from the user's immediate view. This is clear from the abstract to Chamberlin[72], which introduced SEQUEL, and which is quoted in part in Section 6.2. A brief outline of the development of SQL is also included in this chapter.

6.1. DSL Alpha

6.1.1. Retrieval

The form of a basic query in Alpha is

```
GET w [(quota)] (target_list) [: [predicate] [ordering]]
```

to retrieve into the named workspace w those tuples satisfying the predicate expression. A workspace is much the same as a view in SQL, and may be used in subsequent queries just as if it were a relation in the database, although it has no permanent existence. (Note that this removes one of the problems in relational systems, viz. that assignment is only possible to an existing relation. Here the user may create temporary relations in his own workspace; if it is required to retain the result of an expression more permanently, tuples must be moved from the workspace to an existing relation by one of the update commands described below. This is the approach adopted by MURDER, as described in Chapter 3.)

The optional quota modifier specifies that evaluation of the expression is to be terminated as soon as the required number of tuples has been found; thus

```
GET WS (1) (SP.SNUM) : SP.PNUM = "P1"
```

would find <u>one</u> supplier of part P1. This feature has not been retained in SQL.

The result of an expression may optionally be ordered on a nominated attribute by appending

```
UP | DOWN att_name [, att_name] ...
   [ UP | DOWN att_name [, att_name]... ] ...
```

to the expression. Although in theory relations consist of <u>unordered</u> sets of tuples, much data processing depends on

an ordering. SQL retained this facility in the ORDER BY clause.

Both the target list and predicate (qualification) may include functions: COUNT (the number of tuples in a relation), the TOTAL (equivalent of SUM), MAX, MIN and AVERAGE of an attribute, and two new ones, the boolean functions TOP and BOTTOM, with

```
TOP(n, att_name)
```

giving the value true if the current tuple contains for the named attribute the value nth-largest in the set of the attribute values arranged in order. BOTTOM is defined similarly, mutatis mutandis; thus both of

```
GET WS (1) (SPJQ.SNUM) : DOWN SPJQ.QTY
GET WS (SPJQ.SNUM) : TOP(1, SPJQ.QTY)
```

might return supplier numbers corresponding to the maximum QTY value in the SPJQ relation, although where the former always gives only one such value the latter could return more than one. (It is not clear from the paper whether the former is even valid, i.e. whether the presence of the ordering postpones the quota facility until tuples are returned from the sort procedure.) Neither TOP nor BOTTOM were retained in SQL.

Also provided were the so-called image functions. (The name derives from the term 'image set' in the definition of the algebraic quotient operator, defined in Section 4.8, q.v.) The expression

```
GET WS (SPJ.SNUM, ICOUNT(SPJ, SNUM, PNUM))
```

would return (SNUM, COUNT_PNUM) pairs in which the second value is the number of parts supplied by the supplier indicated by the first value. For each 'prefix' SNUM-value, the image set is the set of associated 'suffixes'; this is projected over PNUM (to remove duplicates) and the result counted. SQL provided an alternative - and more clearly expressed - means of achieving the same ends, described in Section 6.3.3.

6.1.2. Updates

Until now the concern has been with identifying tuples which meet certain requirements, although it has been mentioned that this is only part of the business of a Data Manipulation Language. Alpha provided a number of commands which supported modification of stored data.

The HOLD command is like GET, but with two differences:

- the expression is restricted to one relation (compare the rules for updating via a view in SQL); and
- the DBMS is notified of the user's intention to update the indicated relation, and may take steps to prevent problems which might arise from concurrent access.

Once selected tuples had been obtained in the workspace, they could be manipulated (through host language statements - Alpha assumed the presence of a host language for

manipulating attribute values), and then either returned to the base relation using UPDATE, or the updates could be discarded by RELEASE. There was also the PUT command to insert tuples from the workspace into a nominated relation (rather like the inverse of GET, or the INSERT command of MURDER and SQL).

Tuples were removed from a relation by the DELETE command. To remove reference to part P1 from the PART relation would require

```
GET WS (PART) : PART.PNUM = "P1"
DELETE (PART) WS
```

and an unqualified expression such as

```
DELETE (PART)
```

would remove all tuples from the PART relation, although leaving the header information in the database. Note that the set operations can now be obtained by

```
difference:   GET WS (Q)
              DELETE (P) WS

intersection: GET WS1 (P)
              GET WS2 (Q)
              DELETE (WS1) WS2
              DELETE (P) WS1

union:        GET WS (Q)
              PUT (P) WS
```

assuming that P and Q are union-compatible relations. Intersection is here based on the primitive definition

```
P - (P - Q)
```

and union assumes the Alpha convention of elimination of duplicates.

If it was required to remove all reference to a relation, i.e. delete any existing tuples as well as the header definition, the command

```
DROP rel_name
```

was provided. SQL and MURDER retain this distinction between DELETE and DROP.

6.1.3. Other Facilities

Since much data processing is of a record-by-record nature, Alpha provided 'piped mode' using OPEN and CLOSE. Once

```
OPEN (SPJ)
```

had been executed, subsequent GET or HOLD commands on the SPJ relation would load only one tuple into the workspace. This tuple could then be modified or deleted as above, and the cycle repeated until a CLOSE command was executed for the relation. In SQL any such record-by-record processing must be done from a host program.

It was also suggested that it be possible to specify quantifiers at the time a range variable was declared:

```
RANGE JOB j ALL
RANGE SPJ y SOME
GET WS (SPJ.SNUM) : SPJ.SNUM = y.SNUM & j.JNUM = y.JNUM
```

is the equivalent of

```
RANGE JOB j
RANGE SPJ y
GET WS (SPJ.SNUM) : ∀j ∃y(SPJ.SNUM = y.SNUM &
                          j.JNUM = y.JNUM)
```

as a solution to the query "Find who supplies all jobs", again at the expense of separating from an expression information crucial to its meaning. (QUEL goes further, by having range variables implicitly quantified by the existential quantifier if required, i.e. when they appear for the first time in the predicate! This is only possible because QUEL does not have the universal quantifier, and a query in which 'for all' appears must be recast, usually using an intrinsic function and negation. This query could become "Find suppliers for whom the count of jobs which they don't supply is zero". Recent versions of Ingres use the ANY function, with

```
... ANY( expr ) = 1
... ANY( expr ) = 0
```

corresponding to EXISTS and NOT EXISTS respectively.) In SEQUEL no quantifiers were explicitly provided, although a similar result could be achieved, using nested IN queries - see Section 6.3.2. However, IBM's dialect SQL/DS includes EXISTS and NOT EXISTS, with the latter doing the work of the universal quantifier - "for all x, p(x) is true" is equivalent to "there does not exist an x such that p(x) is not true".

6.2. SEQUEL

The language SEQUEL (Structured English Query Language) was first introduced in Chamberlin[74]. At this stage it was just that - a query language. In the abstract of their paper the authors wrote

> "Without resorting to the concepts of bound variables and quantifiers SEQUEL identifies a set of simple operations on tabular structures, which can be shown to be of power equivalent to the first order predicate calculus. A SEQUEL user is presented with a consistent set of keyword English templates which reflect how users use tables to obtain information. Moreover, the SEQUEL user is able to compose these basic templates in a structured manner in order to form more complex queries."

SEQUEL is a small language, as reference to the syntax in Appendix B will show. Indeed it is remarkably small to be able to support such claims.

The simple query "Find who can supply part P1" illustrates the basic mapping:

```
SELECT SNUM
FROM SP
WHERE PNUM = "P1"
```

To find the names of these suppliers two mappings could be composed, as

```
SELECT SNAME
FROM SUPPLIER
WHERE SNUM = SELECT SNUM
             FROM SP
             WHERE PNUM = "P1"
```

Alternatively, two select lists could perhaps be concatenated:

```
SELECT SNAME
FROM SUPPLIER, SP
WHERE SUPPLIER.SNUM = SP.SNUM
  AND PNUM = "P1"
```

although while allowed by the syntax, this may not be meant to be possible. (The

```
SELECT select_list FROM
```

part may be omitted when all attributes are to be retained, which is not the case with the SP relation in this expression.)

The first of these illustrates the notational overloading of SEQUEL. The inner block returns a table containing a single column, the set of SNUM values for suppliers of part P1 as before. For each tuple from SUPPLIER, the SNAME value is retained if its SNUM value matches any value in this set. Thus the sign '=' corresponds to the set membership operator '∈'. That there is an implicit existential quantifier applied to the bound variable SP is hidden from the user, as can be seen by comparing the calculus solution

```
(s{SUPPLIER): ]x{SP(s.SNUM = x.SNUM & x.PNUM = "P1")
```

To find the names of suppliers who do not supply this part requires

```
SELECT SNAME
FROM SUPPLIER
WHERE SNUM ≠ ALL SELECT SNUM
                 FROM SP
                 WHERE PNUM = "P1"
```

It now appears that '=' has the default modifier 'some' (made explicit in later versions as 'ANY'). Compare

```
  (s{SUPPLIER.SNAME):^]x{SP(s.SNUM=x.SNUM & x.PNUM="P1")
= (s{SUPPLIER.SNAME): @x{SP(s.SNUM=x.SNUM => x.PNUM<>"P1")
```

Finding suppliers of more than one part illustrates qualification:

```
SPA: SELECT SNUM
     FROM SP
     WHERE SNUM = SELECT SNUM
                  FROM SP
                  WHERE SPA.SNUM = SNUM
                    AND SPA.PNUM ≠ PNUM
```

Note that labelling blocks means that it was not possible to join a table with itself. However, since a select clause reduces in its simplest form to a table name, it was possible to do operations on tables directly:

```
A   B
```

gave the union of A with B, provided that they were union-compatible. (This simplicity was lost in later versions.)

A table could be 'partitioned' by the values of one or more attributes. To find the quantity of each part supplied required

```
SELECT PNUM, SUM(QTY)
FROM SPJQ GROUP BY PNUM
```

and to find who supplies at least 10 of part P1

```
SELECT SNUM
FROM SPJQ GROUP BY SNUM
WHERE PNUM = "P1"
  AND SUM(QTY) ≥ "10"
```

Here the first predicate applied to individual tuples, while the second applied to each group (because of the function - again later versions used a more explicit form to distinguish between conditions applied to tuples and groups).

Duplicates were removed unless a function was used, although the user could over-ride this default.

Because of the limitations of most keyboards, some of the notation was replaced by further English keywords.

6.3. SEQUEL2 and SQL

The language SEQUEL was taught experimentally to groups of university undergraduates, both with and without computing experience, in an attempt to identify features which inhibited learning (Reisner[76]). As a result, a modified version was designed which also incorporated the results of the author's research in extending the language functionality to cover full data definition, manipulation and control. This version was known as SEQUEL2 (Chamberlin[76]; Appendix C gives the query part of SEQUEL2 syntax.) Since then

- because of potential confusion with another series of products in the market, the name has been changed to SQL (still pronounced 'sequel', derived from Struc-

tured Query Language and which is now a misnomer);

- SQL has been released in a variety of dialects by a number of manufacturers as an interface to a range of systems; and

- there has been a move towards producing a standard Relational Database Language, based on SQL (X3H2[83], [85]).

At Massey University there has always been a strong belief that

> "there are some things which you have to do in order to learn how to do them."

The relational system already in use gave students experience in using a fairly normal algebraic language, and a calculus which - though expressed in a formal notation - preserved the flavour of this approach, and also gave them experience in the concepts of first order predicate calculus. The move towards a standard based on SQL was the reason work was begun towards providing an interface which would give the students experience of this also.

Some research has been done on translating SQL expressions into algebra, presumably because it was felt that algebra is capable of more efficient implementation (Ceri[85]). It was decided instead to adapt the current calculus, for two reasons:

- SQL is basically a calculus; and
- the calculus already used is reasonably efficient.

The main use of the system at Massey has been addressed to teaching better design through experience of data retrieval, and accordingly a subset of the SQL query features was designed.

For the rest of this section the distinction between SEQUEL2 and SQL is made only when there is a significant difference between them.

6.3.1. Simple Queries

The basis of a SQL query is the 'mapping'

```
SELECT select_list
FROM from_list
[WHERE boolean]
```

Finding the cities in which suppliers are located could involve 'projecting' the SUPPLIER relation, by

```
SELECT CITY
FROM SUPPLIER
```

although this does not remove duplicates - SQL removes duplicates either

- on request, or
- whenever such elimination of duplicates is implicit in the query. (This is further expanded below.)

  ```
  SELECT DISTINCT CITY ...
  ```

  forces the elimination of duplicates. (Early specifications for SEQUEL used UNIQUE instead of DISTINCT. MURDER eliminates duplicates as a matter of course,

and so does not use either.)

To find all details about suppliers located in London, 'select' as

```
SELECT *
FROM SUPPLIER
WHERE CITY = "LONDON"
```

but all tuples in the result contain the same value for the CITY attribute. (It is no longer possible, as in SEQUEL, to omit 'SELECT ... FROM' when all attributes are required.) This redundant information can be eliminated by combining selection and projection:

```
SELECT SNUM, SNAME
FROM SUPPLIER
WHERE CITY = "LONDON"
```

A cartesian product and theta-join are achieved simply by

```
SELECT *
FROM SUPPLIER, JOB
```

```
SELECT *
FROM SUPPLIER, JOB
WHERE SUPPLIER.CITY = JOB.CITY
```

respectively, where in the latter case attributes in the WHERE boolean are distinguished by qualification by the name of the parent table. SQL uses table for relation; attributes are termed columns. A natural join could be written

```
SELECT SUPPLIER.*, JOB.JNUM, JOB.JNAME
FROM SUPPLIER, JOB
WHERE SUPPLIER.CITY = JOB.CITY
```

which also illustrates qualification of attributes in the

select list, although here it is not necessary for the last two since they are unambiguous.)

When the same table is used more than once, aliasing is possible:

```
SELECT X.SNUM
FROM SPJ X, SPJ Y
WHERE X.SNUM = Y.SNUM
  AND X.PNUM NOT = Y.PNUM
```

returns supplier numbers of suppliers who supply more than one part. (This query also shows that there is no requirement to retain at least one attribute from each relation mentioned in the from list, a point which has implications for the naive user - see Section 7.2.1.)

If P and Q are two union-compatible relations, the basic set operations are provided as

```
(SELECT *
 FROM P)
 MINUS
(SELECT *
 FROM Q)
```

with UNION and INTERSECTION also being available. Note that these operators now operate on select blocks, not on relations. IBM's SQL/DS includes only UNION directly, but MURDER retains all three set operators.

There is no simple equivalent of the quotient operator, although the effect can be obtained using grouping, as described in Section 6.3.3.

The result of a SQL expression can be obtained in a desired order by appending

```
ORDER BY att_name ASC|DESC [att_name ASC|DESC] ...
```

to the expression, but as MURDER always returns ordered relations this is ignored.

SQL retains the same functions as Alpha (except TOP and BOTTOM) with TOTAL renamed SUM. The number of parts supplied by supplier S2 is found by

```
SELECT COUNT(DISTINCT PNUM)
FROM SP
WHERE SNUM = "S2"
```

Here, DISTINCT is required - without it the result would be the same as for

```
SELECT COUNT(*)
FROM SP
WHERE SNUM = "S2"
```

which simply counts the appropriate tuples.

6.3.2. Nested Queries

Listing all the parts supplied by London-based suppliers could use either of

```
SELECT PNUM
FROM SP, SUPPLIER
WHERE SP.SNUM = SUPPLIER.SNUM
  AND CITY = "LONDON"

SELECT PNUM
FROM SP
WHERE SNUM IS IN SELECT SNUM
                 FROM SUPPLIER
                 WHERE CITY = "LONDON"
```

In the latter case, 'IS IN' more or less corresponds to the set-membership operator 'Є', i.e. the nested query may be seen as returning a set of SNUM values. Alternatively it can be viewed as a disguised existential quantifier; the calculus for this query would be

(x{SP.PNUM) :]s{SUPPLIER (x.SNUM=s.SNUM & s.CITY="LONDON")

In either case note that the nested block is constrained by the context to return a table with an externally defined number of columns respectively based on suitable domains.

A discussion of the negated form 'IS NOT IN', and its mapping on to the universal quantifier, is given in Section 7.2.3.

In SEQUEL either operand of the IN function could be a literal, and more than one column could be referenced. An expression to find any suppliers who supply a part to a job to which S1 supplies the same part could be

```
SELECT SNUM
FROM SPJ X
WHERE SNUM NOT = "S1"
  AND <PNUM, JNUM> IS IN SELECT PNUM, JNUM
                         FROM SPJ
                         WHERE SNUM = "S1"
```

in which the angle-brackets act as a tuple-constructor function. MURDER retains the facility, but without the notation. Again, as an example of the use of literals, finding suppliers of any of parts P1, P2 and P5 could be written

```
SELECT SNUM
FROM SP
WHERE PNUM IS IN "P1", "P2", "P5"
```

MURDER again provides this facility with a modified syntax. However there is the restriction that literals occur only in the right part. This may seem arbitrary, but is consistent with ordinary English usage, which tends to put the subject first in a sentence.

The type of comparison using IN was also - rather unsatisfactorily - offered between sets. Suppliers who don't supply any parts not supplied by supplier S2 could be found by

```
SELECT SNUM
FROM SP X
WHERE (SELECT PNUM
       FROM SP
       WHERE SNUM = X.SNUM)
 IS IN
      (SELECT PNUM
       FROM SP
       WHERE SNUM = "S2")
```

Here 'IS IN' now represents set containment. The last select block returns a table of PNUM values, corresponding to parts supplied by supplier S2. For a given X-tuple (the alias is required here), first obtain the set of PNUM values for the appropriate supplier, and if this set is a subset of the first set, retain the SNUM value from the X-tuple. In the middle block the first reference to SNUM does not require qualification, as it is implicitly qualified by the local instance of SP; the same is true for the SNUM reference in the last block. When set comparison is involved SQL automatically eliminates duplicates, although

this expression should probably be

```
SELECT DISTINCT SNUM ...
```

to eliminate duplicate SNUM values from the final result; an alternative is

```
SELECT SNUM
FROM SUPPLIER
WHERE (SELECT PNUM
       FROM SP
       WHERE SNUM = SUPPLIER.SNUM)
 IS IN
      (SELECT PNUM
       FROM SP
       WHERE SNUM = "S2")
```

which requires neither DISTINCT (since SNUM is the primary key of SUPPLIER) nor aliasing, although the second SNUM reference in the middle block still must be qualified. Available set comparisons in SEQUEL were

```
[NOT] =
IS [NOT] IN
CONTAINS
DOES NOT CONTAIN
```

after the initial mathematical notation was changed, and the distinction of proper set containment dropped. The last two are the inverses of the forms using IN. SEQUEL2 suggested

```
IS [NOT] CONTAINED IN
```

instead of the second form shown above. MURDER allows set comparisons (using the CONTAINED IN form) only with the SET function, described in the next section.

6.3.3. Grouping

Section 6.1.1 used an Alpha image function to find how many parts each supplier currently supplies. The SQL solution to this query could be expressed

```
SELECT SNUM, COUNT(DISTINCT PNUM)
FROM SP
GROUP BY SNUM
```

(Note that in SEQUEL the GROUP clause is appended to the FROM clause; in SEQUEL2 and SQL it follows the WHERE clause, i.e. if this query had a WHERE clause it would now precede GROUP.) Tuples from relation SP are grouped according to SNUM value; each group is projected over PNUM (removing duplicates), and the count of the result returned along with the appropriate SNUM value. In general only properties of the group may be selected.

Additionally, a query may include a HAVING clause, which is to a group what the WHERE clause is to individual tuples. The previous query about suppliers who do not supply anything S2 does not supply becomes

```
SELECT SNUM
FROM SP
GROUP BY SNUM
   HAVING SET(PNUM) IS CONTAINED IN SELECT PNUM
                                    FROM SP
                                    WHERE SNUM = "S2"
```

The SET function constructs a set, here of PNUM values for a particular supplier. This set is then compared with the table returned by the nested query.

Suppliers of more than 1 part could be found directly by

```
SELECT SNUM
FROM SP
GROUP BY SNUM
   HAVING COUNT(DISTINCT PNUM) > 1
```

This now distinguishes between conditions applied to tuples and groups.

It is possible to simulate division. To find suppliers of the same part to all jobs, the algebra solution given in Chapter 4 was

```
PROJ [DIV SPJ
      BY [PROJ JOB OVER JNUM]]
OVER SNUM
```

The SQL equivalent could be

```
SELECT SNUM
FROM SPJ
GROUP BY SNUM, PNUM
   HAVING SET(JNUM) = SELECT JNUM
                      FROM JOB
```

which also illustrates grouping on more than one attribute.

In the condition of a HAVING clause the left part must be a function evaluating a property of the group. (Again this restriction is justified by appealing to normal English usage.) The right part can be anything which can be evaluated for the group, either a literal, another function, or a further nested query. In this case the nested query is further constrained to having only one row (per group), i.e. to producing a 'scalar' result. Note

that the explicit grouping of SQL allows the same functions to be used in both the select list and the HAVING clause, where Alpha provided separate image functions. The SET function illustrated above is available only in a HAVING clause.

6.3.4. Indexes

In SEQUEL2 two data structures were suggested to support data access, viz. images and links. The former provided an index to a relation on specified attributes, and indeed are now called indexes in SQL. One could

```
CREATE [UNIQUE] [CLUSTERING] IMAGE image_name
    ON relation_name ( ordering )
```

If specified, UNIQUE (or DISTINCT) indicated that the attribute(s) named in the ordering constituted a candidate key, and a tuple could not be stored or modified if the value of an existing key would be duplicated. There was, however, no indication of primary or alternate keys.

At most one index could have the CLUSTERING property. In this case, the DBMS would attempt to store tuples in physically contiguous locations in index order, to improve access efficiency.

A link is

> "a set of pointers that connect the tuples of one relation to the tuples of another relation which match according to a certain attribute"

i.e. the database was to contain a physical representation

of an equijoin. One could, for example

```
CREATE [CLUSTERING] LINK SUPP-SP
  FROM SUPPLIER (SNUM)
    TO SP (SNUM)
  [ORDER BY PNUM ASC]
```

if joining SUPPLIER and SP over the SNUM attribute is a common operation, which it is, and additionally specify if required that the link is to be maintained in ascending order of associated PNUM values, or whatever order is required. Again the CLUSTERING property is a request to aid access by attempting to locate associated tuples in contiguous physical locations. Released versions of SQL do not support links.

Note that this is not a foreign key mechanism, for two reasons:

- it is the wrong way round, in the sense that the SP tuples should be associated with SUPPLIER tuples - it could be declared the other way round, but in this case each SP tuple would have a (single) pointer to the corresponding SUPPLIER tuple; and

- there is no provision for ensuring the existence of an associated tuple in the target relation.

It is important to note that while the user could CREATE and DROP indexes and links, he had no other control over them, and in particular could not reference them in a query. All queries continue to be expressed in non-procedural terms; but the DBMS could take advantage of the

existence of indexes and links in choosing efficient strategies for obtaining their solutions.

As stated in Chapter 3, MURDER does not support indexes because they were incompatible with the storage strategy adopted for the small data-volumes envisaged in a teaching environment.

6.3.5. Views

Reference has already been made to the problems of relational assignment and temporary relations. Alpha provided each user with a workspace, in which he was free to create temporary relations by assignment, although these relations had no permanent existence. In SQL the nearest corresponding feature is the view, although this does not have the same purpose. A view is created by the command

```
DEFINE VIEW view_name [ ( att_name_list ) ]
   AS sql_expr
```

where the attribute name list may be omitted if the attributes in the result of the defining expression are unambiguous. The expression may reference more than one base table. In this way the user can be given a tailored view of data, by eliminating attributes which are not relevant, or to which the user should not be permitted access.

There is another important difference, in that a 'view' provides a dynamic window into one or more relations. Once defined, a view may be used in the same way as a 'base table', except that updates are not permitted in

certain circumstances, e.g. via a view derived from more than one relation. (This is because of possible problems resulting from the projection out of a primary key attribute from one or more relations in the view definition.) Some questions remain to be answered. For example, if a view is truly dynamic, it would be possible for a tuple to be present in the view when first referenced, but modified or deleted (by another user) before a subsequent attempted reference.

Again, the existence of a DISTINCT INDEX on the base relation could cause problems. If an attempt is made to store a new tuple which has a unique key so far as the view is concerned, but which would duplicate an existing key in the base relation (of which the view user is unaware), at what point does the error occur? And how is the user to avoid this rather puzzling situation?

At present MURDER does not support views, although the system could be modified by maintaining a separate list of macros, each containing a single assignment statement. Reference to a relation would then require scanning this list as well as the normal relation lists. If the relation is found in the latter, it is used; but if in the former, the defining expression would be evaluated and the result placed in the temporary list, and used as before. The scope of such a relation could then be related to the manner of its creation:

- if the macro call has no parameters (which would be the usual case) and the call is itself from a macro, it would persist until the termination of that calling macro;

- otherwise it would persist only for the duration of the current statement.

The result of this modification would still have rather more the flavour of Alpha's workspaces, but this is considered preferable for the reasons already given.

6.3.6. Updates

SEQUEL2 provided three statements for modifying the contents of a relation. One could

```
INSERT INTO rel_name : sql_expr | < value_list >
```

to add either the result of a query expression, or a single tuple, to an existing relation. There was no corresponding means for removing a single tuple, although

```
DELETE rel_name [ where_clause ]
```

would do this if the condition identified a unique row, e.g. by referencing a key. As before, the unqualified form deleted all tuples leaving only the header, and DROP removed all reference to the relation from the database. These first two are tuple-oriented in the sense that they operate on whole tuples; only UPDATE allowed the manipulation of attribute values:

```
UPDATE rel_name [ alias ]
   SET assignment_list [ where_clause ]
```

where

```
assignment_list ::= assignment [, assignment_list]
assignment ::= att_name = expr | ( sql_expr )
```

and the where clause is as described for a query expression. This allowed individual attributes to be manipulated, either in all tuples, or only in tuples satisfying the optional condition. Note that the sql_expr in the last case is constrained to evaluate to a scalar 'relation', i.e. with exactly one column and one row.

In addition to the normal relation definition, relations could be created by assignment:

```
ASSIGN TO rel_name [ ( att_name_list ) ] : sql_expr
```

This statement allows the new relation to inherit attribute names from the defining expression, or optionally they can be specified by the user, e.g. to avoid ambiguity (the name list is required in this case) or when more appropriate names are desired. The resulting relation assumes a full role in the database, i.e. this facility goes further than the workspace in Alpha, and is only possible because SEQUEL2 knew nothing about keys. However released versions of SQL do not seem to have included assignment, presumably because of these problems, discussed in Chapter 2.

CHAPTER 7

MURDER SQL

This chapter presents the MURDER implementation of SQL, describing - and commenting on - each feature as it is introduced. All the features described in Section 6.3 are included except expressions in the select list, and the update statement is also provided.

7.1. The Basic SQL Expression

The central idea of SQL is that of a 'mapping', expressed as

```
 SELECT select_list
 FROM from_list
[WHERE boolean]
```

Discussion of the optional WHERE clause is deferred to the next section. The simplest form is equivalent to an algebraic projection. The following SQL/calculus pairs of expressions are equivalent:

```
SELECT SNAME
FROM SUPPLIER

(s{SUPPLIER.SNAME)
```

```
SELECT SNUM, SNAME
FROM SUPPLIER

(s{SUPPLIER.SNUM, s.SNUM)
```

```
SELECT *
FROM SUPPLIER

(s{SUPPLIER)
```

Note that

- SQL does not require qualification of attribute names, although this is allowed; and

- SQL uses the asterisk '*' when all attributes are intended. (Calculus simply references the relation.)

The '*' has an extended use in SQL, illustrated by the following equivalent forms of a product

```
    SELECT SUPPLIER.*, JOB.*
    FROM SUPPLIER, JOB
```

```
or  SELECT *
    FROM SUPPLIER, JOB
```

cf. (s{SUPPLIER, j{JOB)

and of a projection of a product

```
    SELECT SNAME, JNAME
    FROM SUPPLIER, JOB
```

```
or  SELECT SUPPLIER.SNAME, JOB.JNAME
    FROM SUPPLIER, JOB
```

cf. (s.SUPPLIER.SNAME, j{JOB.JNAME)

```
    SELECT SUPPLIER.CITY, JOB.*
    FROM SUPPLIER, JOB
```

cf. (s{SUPPLIER.CITY, j{JOB)

The partly qualified alternative for the last example

```
    SELECT SUPPLIER.CITY, *
    FROM SUPPLIER, JOB
```

is either not legal (some SQLs insist that an unqualified '*' be the only token in the select list - MURDER takes

this approach) or means something different (the seven attributes, SUPPLIER.CITY plus the six from the product of SUPPLIER and JOB); and

```
SELECT CITY, JOB.*
FROM SUPPLIER, JOB
```

may not be an alternative, depending on how the ambiguity in the CITY attribute is resolved. In MURDER unqualified attribute references are resolved by working from the most recently declared range forwards. In this case CITY is taken as a reference to JOB.CITY; in fact this form too should not be legal - an attempt to write a calculus expression of the qualified form it represents

```
SELECT JOB.CITY, JOB.*
FROM SUPPLIER, JOB
```

fails, for there is no way of introducing reference to the SUPPLIER relation (from which no attributes are being retained) into the target list. (This point is further considered in the next section.)

The implementation of SQL to this point differs from the calculus only in the parsing procedure. In particular the SELECT ... FROM ... format means that the select list has to be saved until the from list has been parsed, where for the calculus the specification of the appropriate range at the point of introduction of each new variable allows a one-pass approach. As a consequence errors in SQL are often detected late - an error detected in the select list may be caused by a mistake in the from list -, making

error-recovery more difficult. (In MURDER if the mistake is at the point the error is detected, the user is able to make the correction, otherwise the expression must be aborted. Could not the mapping be expressed FROM ... SELECT ..., putting the declarations first?)

SQL also allows aliasing, illustrated (though not very meaningfully) by the example

```
SELECT X.SNUM, Y.PNUM
FROM SPJ X, SPJ Y
```

The alias is used as in the varname field of the corresponding calculusnode. As a consequence of this, a qualified attribute reference in MURDER must use the alias - SQL does not seem to insist on this. Aliasing is only required when it is necessary to distinguish between two instances of the same relation. Note that according to the definition for resolving attribute references, each of the following potentially dangerous forms means the same:

(1)
```
SELECT X.SNUM, PNUM
FROM SPJ X, SPJ
```

The ambiguity is resolved as described above; however a further problem is that, if the two instances of the same relation occur at different levels in the expression (as often happens), and if the former was not aliased at the time, it is now too late.

(2)
```
SELECT X.SNUM, SPJ.PNUM
FROM SPJ X, SPJ Y
```

This illustrates the observation that there seems to be no requirement to use an alias when it exists.

(3)
```
SELECT X.SNUM, SPJ.PNUM
FROM SPJ X, SPJ
```

If there is no requirement to use an alias then this, like (1), is only unambiguous because of the reference resolution rules.

(4)
```
SELECT X.SNUM, SPJ.PNUM
FROM SPJ X, SPJ Y
```

This is a more blatant example of ignoring an available alias.

In contrast to all these, the approach taken by the calculus, assigning a unique range variable to each range, is clearly preferable to the illusory convenience of SQL. An alias should effectively replace the relation name for the duration of the expression, as is the case in MURDER.

The next feature of SQL included (in a slightly constrained manner) is illustrated by

```
SELECT MAX(WEIGHT) FROM PART
```

for which MURDER provides no calculus equivalent (although DSL-alpha did - see Section 6.1). The functions available for numeric data are AVG (average), SUM, MAX, MIN. AVG always returns a real, the others the same type as the attribute to which they are applied. COUNT(att_name) (i.e. unique values of the attribute) and COUNT(*) (which

merely counts the tuples in the universe) may be used for any data type, and return an integer. Since MURDER removes duplicates, a query including SUM or AVG must be temporarily expanded to include all attributes from the target range. The resulting attribute is named COUNT (for COUNT(*)) or is formed by prefixing an appropriate 4-character string to (the first 7 characters of) the named attribute, e.g. in this example MAX_WEIGHT.

COUNT(*) simply returns the tuple-count field from the relation node. COUNT(att_name) requires projection over the named attribute first (to remove duplicates) before this is done.

MAX (or MIN) could be found trivially if an index on the named attribute exists; otherwise a single pass through the relation in first, next order is needed, saving the first value and updating whenever a greater (lesser) value is met.

AVG and SUM similarly require a pass through the relation accumulating the value of the attribute from each tuple, and in the former case dividing by the tuple-count (found as for COUNT(*)) before returning.

In this simple form of a SQL query MURDER requires that the select list be either all attribute references or all function references:

```
select_list ::= * |
                att_ref_list |
                fn_ref_list
```

```
att_ref_list ::= att_ref [, att_ref_list]

fn_ref_list ::= fn_ref [, fn_ref_list]

fn_ref ::= COUNT(*) |
           function ( att_ref )

att_ref ::= [qualifier .] att_name

function ::= COUNT |
             AVG |
             SUM |
             MAX |
             MIN

from_list ::= from_primary [, from_list]

from_primary ::= relation_name [alias] |
                 '[' sql_expr ']' alias

qualifier ::= relation_name |
              alias
```

As was done for the calculus, MURDER extends the syntax to allow recursion by including a SQL expression enclosed in brackets in the from list. Note that in this case an alias is required.

A feature of SQL not currently supported is the ability to include expressions in the select list, illustrated by

```
SELECT QTY * PRICE
FROM ORDER, STOCK
WHERE ORDER.PNUM = STOCK.PNUM
```

(although MURDER does provide this facility in another form - see the discussion of MAKE in Section 3.2).

7.2. The WHERE Clause

```
sql_expr ::= sql_block

sql_block ::= SELECT select_list
              FROM from_list
              WHERE boolean
```

The WHERE clause identifies which tuples from the universe participate in the result. A variety of the suggested forms for boolean are provided, discussed separately in the following paragraphs. The major forms not supported are

```
expr BETWEEN expr AND expr
```

which can always be written as two booleans, and the use of the SET function outside the HAVING clause. (See Section 7.3.2)

7.2.1. Simple Booleans

The simple forms of boolean are similar to the select expression described for algebra, and to simple qualification for calculus:

```
boolean ::= NOT boolean |
            boolean AND boolean |
            boolean OR boolean |
            IF boolean THEN boolean |
            ( boolean ) |
            comparison

comparison ::= left_part relop right_part

left_part ::= att_ref

right_part ::= att_ref |
               function ( att_ref ) |
               literal
```

Thus, suppliers of part P1 may be found by

```
SELECT SNUM
FROM SPJ
WHERE PNUM = "P1"
```

cf. (x{SPJ.SNUM): x.PNUM = "P1"

or, if supplier names are required, by

```
SELECT SNAME
FROM SUPPLIER, SPJ
WHERE SUPPLIER.SNUM = SPJ.SNUM
          AND PNUM = "P1"
```

cf. (S{SUPPLIER.SNAME):]X{SPJ(S.SNUM=X.SNUM & X.PNUM="P1")

This again illustrates the major difference between SQL and calculus already mentioned: in calculus it is not possible to include a range in the target list without retaining at least one of its attributes; in SQL it is possible, and even encouraged. (Date[84] gives this as an example ...) The calculus equivalent given above makes the existential quantifier governing such ranges explicit, where in the SQL version the presence of the quantifier might not be suspected. The attempted negation

```
SELECT SNAME
FROM SUPPLIER, SPJ
WHERE NOT (SUPPLIER.SNUM = SPJ.SNUM
               AND PNUM = "P1")
```

gives in general a list of all supplier names, being equivalent to

(s{SUPPLIER.SNAME):]x{SPJ(s.SNUM<>x.SNUM | x.PNUM<>"P1")

instead of the required

(s{SUPPLIER.SNAME): ^]x{SPJ(s.SNUM=x.SNUM & x.PNUM="P1")

The problem could be avoided by requiring that at least one attribute from each range in the from list be referenced in the select list.

The equivalent of a theta-join is illustrated by

```
SELECT *
FROM SUPPLIER, SPJ
WHERE SUPPLIER.SNUM = SPJ.SNUM
```

cf. (s{SUPPLIER, x{SPJ): s.SNUM=x.SNUM

7.2.2. EXISTS Sub-query

```
boolean ::= [NOT] EXISTS sql_block
```

Note that the original SEQUEL did not explicitly include quantifiers, using instead a form of the feature described in the next section; EXISTS is described first, however, because the calculus already covered will have made it familiar.

Note that only EXISTS and NOT EXISTS are available in SQL/DS. The supplier-name query from the preceding section could be expressed

```
SELECT SNAME
FROM SUPPLIER
WHERE EXISTS SELECT *
             FROM SPJ
             WHERE SUPPLIER.SNUM = SNUM
               AND           PNUM = "P1"
```

making explicit the coupling of an existential quantifier to the range SPJ. Note that only the reference to the supplier's SNUM attribute requires qualifying to distinguish between the external and local instances of attributes with this name - the rule given above for resolution of references ensures that the references to local SPJ attributes are interpreted correctly, although this casual

approach could lead to problems.

When more than one instance of the same base table appears, all but the last require aliasing; in the query regarding suppliers of more than one part we could give

```
SELECT SNUM
FROM SPJ X
WHERE EXISTS SELECT *
             FROM SPJ
             WHERE X.SNUM = SNUM
               AND X.PNUM NOT = PNUM
```

Both these examples can be negated correctly simply by inserting NOT before EXISTS. (MURDER will also accept '<>' for the usual SQL form 'NOT =' .)

As with calculus, EXISTS is handled by recursive calls on a function which parses a sql_block. This function maintains the list of calculusnodes as required, and returns a pointer to the root of the expression sub-tree built for the qualifying boolean. Parameters to this function initially included the representation of the appropriate quantifier, and a flag indicating whether the expression needed negating - recall that the calculus evaluation algorithm only knows about ']' and '@' quantifiers, and hence must use the equivalence

```
^]x(p) = @x(^p)
```

(This turned out to be the wrong approach, as explained in the next section.) Sub-queries can be nested in this way to an arbitrary depth in the implementation of MURDER.

As with calculus, queries may be built up step by step from the inside: knowing supplier numbers for suppliers of more than one part, the expression can be extended to return supplier names by adding a 'join':

```
SELECT SNAME
FROM SUPPLIER
WHERE EXISTS SELECT *
             FROM SPJ X
             WHERE SUPPLIER.SNUM = SNUM
               AND EXISTS SELECT *
                          FROM SPJ
                          WHERE X.SNUM = SNUM
                            AND X.PNUM NOT = PNUM
```

which is the exact equivalent of the calculus solution:

```
(s{SUPPLIER.SNAME): ]x{SPJ(s.SNUM=x.SNUM &
                    ]y{SPJ(x.SNUM=y.SNUM & x.PNUM<>y.PNUM))
```

Turning now to the query about jobs which could be supplied entirely by supplier S2, SQL has

```
SELECT JNAME
FROM JOB
WHERE NOT EXISTS SELECT *
                 FROM SPJ
                 WHERE JOB.JNUM = JNUM
                   AND NOT EXISTS SELECT *
                                  FROM SP
                                  WHERE SPJ.PNUM = PNUM
                                    AND SNUM = "S2"
```

Translating into calculus, and transforming the resulting qualification, gives successively

```
^]x{SPJ(j.JNUM=x.JNUM & ^]y{SP(x.PNUM=y.PNUM & y.SNUM="S2"))
 @x{SPJ(j.JNUM<>x.JNUM | ]y{SP(x.PNUM=y.PNUM & y.SNUM="S2"))
```

which is the same as the calculus solution, and the same data-structure representation is developed. Note that the

sequence of calls to the sql_block function uses parameters (the first parameter is the recursion level)

```
0, ' ', FALSE
1, '@', TRUE
2, ']', FALSE
```

In the first activation the quantifier '^]' is represented by the last two parameters in the second call, viz '@', TRUE. Accordingly, in treating the WHERE clause of the nested mapping in the second activation, ^'=' becomes '<>', ^'AND' becomes 'OR', and ^'NOT EXISTS' becomes 'EXISTS', so that in the final call these parameters are ']', FALSE.

SQL treats EXISTS as a boolean function. From the implementation of the calculus, ']' becomes a projection of a selection of a product; hence the truth/falsity of the boolean function corresponds to the presence/absence of a tuple in the result.

7.2.3. IN Sub-query

In its simplest form

```
boolean ::= att_ref_list IS [NOT] IN '[' tuple_list ']'
tuple_list ::= tuple [, tuple_list]
tuple ::= < value [, value_list] >
```

the IN query may be regarded as simply a shorthand for the disjunction of simple booleans:

```
SELECT SNUM
FROM SPJ
WHERE PNUM = "P1"
   OR PNUM = "P2"
```

may be expressed

```
SELECT SNUM
FROM SPJ
WHERE PNUM IS IN [<"P1">, <"P2">]
```

This notation is not in SQL, although the earlier dialect SEQUEL2 used the angle brackets '<' and '>' as a tuple constructor function. In fact the query "Who supplies part P1?" may be written

```
SELECT SNUM
FROM SPJ
WHERE PNUM IS IN [<"P1">]
```

(In this example 'IS IN' means the same as '=', and the former example can be seen as an alternative way of expressing the 'modified' query form '= ANY'. SQL has modified comparisons

```
att_ref θ [ANY | ALL] sql_block
```

where θ represents any relational operator. This form allows the nested query to return a table, a set of values. If ANY is used the current tuple participates in the result if the named attribute bears the specified relation to at least one element in this set - which is not what the naive user might expect! In non-technical English the user could confuse ANY with ALL, which requires the comparison to be true for each value returned by the nested query. For example

```
... WHERE PNUM NOT = ANY SELECT PNUM ...
```

retains the current tuple if there is at least one different PNUM-value in the result of the nested query, not, as might be expected, if this table contains no equal value. What is required for this condition is

```
... WHERE PNUM NOT = ALL SELECT PNUM ...
```

MURDER does not support the modifiers ANY and ALL.)

For practical purposes the tuple list can be seen as a literal relation, with, for each literal tuple, values drawn from the domain of the corresponding attribute in the preceding attribute list. The implementation proceeds as if the tuple list in this production is replaced by

```
SELECT *
FROM [literal_relation]
```

which is a sort of inverse recursion; if a relation in a from list may be replaced by a bracketed SQL expression, why not replace this simple form of expression by the (value of the) relation? The extreme case would be

```
... WHERE att_ref_list IS IN [relation_name]
```

The more general form of the IN query is

```
boolean ::= att_ref_list IS [NOT] IN sql_block
```

where the attributes in att_list must correspond to those in the nested select list. This provides an alternative means of nesting sub-queries, in a less formal manner than

with EXISTS, and does not necessarily add anything to the power of SQL, although it does increase its expressiveness. (It was one of the means of nesting in the original SEQUEL, when EXISTS did not appear.) Consider again the query about names of suppliers of part P1; this could be expressed

```
SELECT SNAME
FROM SUPPLIER
WHERE SNUM IS IN SELECT SNUM
                 FROM SPJ
                 WHERE PNUM = "P1"
```

Here the inner block appears to return a set of SNUM values, and a tuple from the outer block participates in the result if its SNUM value appears in that set. As for EXISTS, the truth of the IN boolean function is inferred from the presence of a tuple in the result of an implied join over SNUM. This could then be implemented as if this were merely another way of expressing the same thing, with both giving the equivalent calculus:

```
(s{SUPPLIER.SNAME): ]x{SPJ(s.SNUM=x.SNUM & x.PNUM="P1")
```

Indeed, initially the treatment follows these lines. Note however that in this example all attribute references in the nested query are 'local', whereas in the EXISTS version reference is made within the inner query to the 'external' attribute SUPPLIER.SNUM. It follows that the inner query may be evaluated immediately, as if

```
(s{SUPPLIER.SNAME): ]x{[(x{SPJ.SNUM): x.PNUM="P1"]
                        (s.SNUM=x.SNUM)
```

had been written. Such an approach is desirable, as by decreasing the size of the inner range the efficiency of execution is increased.

Achieving this required considerable modification to the algorithm already described. To check the validity of the sub-query an extra parameter is added to the sql_block call. This is a pointer to a list of pnodes in each of which the first pn-value is the attribute index (in the header for the putative product of all ranges so far) of the corresponding attribute referenced in the list preceding IN. Within the inner activation of the sql-block function, as the select list is being parsed, attributes are checked for number and domain against the corresponding attribute indicated by this index, and the second pn-value is set to the appropriate index in the (enlarged) product header. A local variable retains the pointer to the first calculusnode created in the inner query. After building the local expression tree, the minimum offset of any attribute referenced in the qualifying expression is compared with the cumsize field in this node: if it is less, an external attribute has been referenced and the function returns the root of the tree, as described earlier; otherwise the inner query is local and can be evaluated, passing the pointers to the local calculusnode list head and expression tree root into the universe procedure. (Although not mentioned explicitly before, this procedure attaches its final result to the relp pointer of this leading calculusnode, discarding any subsequent nodes, and

updating the global product header.) Before returning, however, the second pn-value in each pnode of the attribute list passed in is updated to the appropriate value in the sequence

```
localcnhead->cumatt + 1, + 2 ...
```

since the first, second ... attributes of the locally-evaluated result are to take part in the implicit join. Now the expression tree is discarded and a null value returned.

In the calling activation of sql_block, the pnodes in the list representing attributes are now used to create (or extend) its expression tree with a representation of the necessary join terms.

For the query "Find details of SPJ-tuples for suppliers other than S2 who supply the same parts to the same jobs as S2 does" the SQL expression

```
SELECT *
FROM SPJ X
WHERE SNUM NOT = "S2"
  AND PNUM, JNUM IS IN SELECT PNUM, JNUM
                       FROM SPJ
                       WHERE SNUM = "S2"
```

is translated to the equivalent of the calculus

```
(x{SPJ): x.SNUM<>"S2" &
         ]y{[(y{SPJ.PNUM, y.JNUM): y.SNUM="S2"]
            (x.PNUM=y.PNUM & x.JNUM=y.JNUM)
```

There seems to be no requirement in SQL that a nested IN query be local as defined above, although ordinary English

semantics would suggest that this is a reasonable restriction. The query above could be expressed

```
SELECT *
FROM SPJ X
WHERE SNUM NOT = "S2"
  AND PNUM IS IN SELECT PNUM
                 FROM SPJ
                 WHERE X.JNUM = JNUM
                   AND   SNUM = "S2"
```

in which the second join term is now explicit in the nested query, so that it is no longer local, and hence cannot be evaluated as described above. The equivalent calculus is

```
(x{SPJ): x.SNUM<>"S2" & ]y{SPJ
            (x.PNUM=y.PNUM & x.JNUM=y.JNUM & y.SNUM="S2")
```

This will reduce after a certain amount of manipulation to the same calculus as was given before, by applying the optimizing procedures described in Chapter 5. It is an unfortunate feature of SQL that the efficiency of evaluation of a query seems to be dependent on the elegance of a user's expression, unless the implementation is capable of the quite sophisticated manipulation needed to reduce an expression to an equivalent simpler form.

In passing it is worth noting that it might be simpler to build the external join part of the expression tree during parsing the nested select list, but in this case the resulting tree would require subsequent modification, just as the attribute list is modified above. If, however, the SQL language required nested IN queries to be local, this could be the preferred approach.

Two problems remain to be discussed, the first being the negation 'NOT IN'. If IN can be treated as an implied ']', NOT IN would become '^]'; names of suppliers who do not supply part P1 would be given by

```
SELECT SNAME
FROM SUPPLIER
WHERE SNUM IS NOT IN SELECT SNUM
                     FROM SPJ
                     WHERE PNUM = "P1"
```

and this introduces the second point: the calculus for this query is

```
 (s{SUPPLIER.SNAME): ^]x{SPJ(s.SNUM=x.SNUM & x.PNUM="P1")

=(s{SUPPLIER.SNAME): ^]x{[(x{SPJ.SNUM): x.PNUM="P1")]
                          (s.SNUM=x.SNUM)

=(s{SUPPLIER.SNAME): @x{[(x{SPJ.SNUM): x.PNUM="P1")]
                          (s.SNUM<>x.SNUM)
```

The point about this sequence of transformations is that, when the quantifier '^]' is met, the next activation of function sql_block is called with parameters '@', TRUE. This is correct if the nested query is non-local (as in the first calculus version above), but if the sub-query can be evaluated locally (last version) the local qualification should not be negated, although the external part, the implicit join, is. Thus, with the approach described above, the local expression tree must first be negated again. The correct approach, then, is to ignore the negate-flag until the inner activation is about to return and then if necessary first negate the local tree.

7.3. The GROUP Clause

```
sql_block ::= SELECT select_list
              FROM from_list
             [WHERE boolean]
             [GROUP BY group_list
                [HAVING having_boolean]]

group_list ::= att_ref_list
```

7.3.1. Simple Grouping

SQL introduces in the GROUP clause a facility already provided in DSL-alpha by the 'image' functions (see Chapter 6) - the ability to partition a relation into sub-relations with the same value for one or more attributes, and to retain in the select list properties of this partition, or group. (Discussion of the optional HAVING clause is deferred until the next paragraph.) First the syntax of the select list is extended:

```
select_list ::= * |
                att_ref_list [, fn_ref_list] |
                fn_ref_list
```

For MURDER there are some additional restrictions imposed by this definition:

- in the absence of a GROUP clause the select list must not contain attribute and function references; and

- in the presence of a GROUP clause both may appear in the select list, with all attribute references before any function reference.

Further restrictions are:

- while a GROUP-query may be nested, it must be local; and

- any attributes referenced in the select list must, while arguments to functions must not, also appear in the group list. (This is a way of stating that in the select list of a GROUP-query, only properties of the group may be SELECTed.)

Using this feature, how many parts are supplied by each supplier can be found by

```
SELECT SNUM, COUNT(PNUM)
FROM SP
GROUP BY SNUM
```

or how many of each part they supply by

```
SELECT SNUM, PNUM, SUM(QTY)
FROM SPJQ
GROUP BY SNUM, PNUM
```

Two representations of the select list are built, one containing only the attributes referenced (whether simply or as the argument to a function) and the other containing pnodes for each attribute or function referenced, with the nodes for the latter including

- an indication that the node represents a function reference;

- for COUNT(att_name) the result field pointing to a relnode for the result a projection over the required

attribute; and in this case

- the lson field pointing to a pnode for the required projection. (See Figure 7.1)

As with non-grouped queries, AVG, SUM and COUNT(*) require the first of these lists to be temporarily expanded to include all attributes so that the elimination of duplicates by MURDER routines will not produce spurious results.

In a query which includes the optional WHERE clause as well as the GROUP clause, the former determines which tuples appear in the local universe to participate in the grouping. In any case, since MURDER requires that a GROUP-query be local, the universe is evaluated at this point.

The grouping is now carried out using Algorithm 7.1. This is not ideal, since it copies part of the universe to a new temporary relation with the same characteristics as the universe, but makes it easier for the facilities already in existence to be used. (A refers to the local universe, B to the current group, and pG is the group-identifying string, which must by now be a prefix, since attributes precede any functions in the select list.)

Procedure do_group traverses the group_ptr list, copying either attributes or the result of the specified function (applied to the result of a projection if required) into the new tuple. Finally, the completed new tuple is added

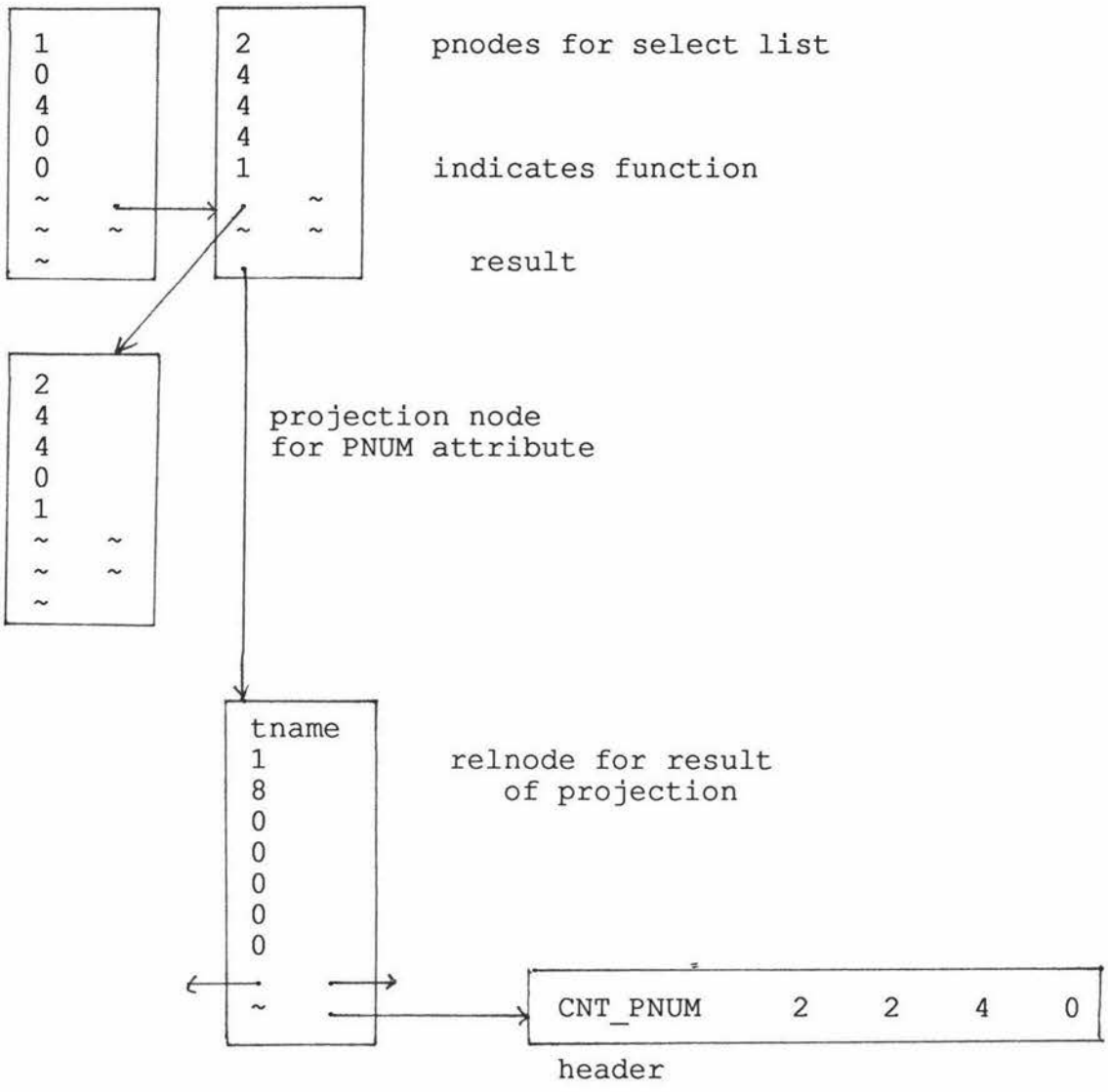


Figure 7.1 - Data Structures for GROUP Query using COUNT

```
more = first(A);
while more
   pG = pA;
   while pG = pA & more
      add(B, tA), more = next(A);
   do_group(group_ptr);
```

Algorithm 7.1 - High Level GROUP Logic

to the result (after a lookup operation, since not all group-identifying attributes need be selected) and the body of the group is discarded.

7.3.2. The HAVING Clause

A GROUP clause may optionally include its own qualifying expression in the HAVING clause, which is to the group what the WHERE clause is to individual tuples, i.e. it is used to determine whether a group participates in the result.

Names of suppliers of more than one part may be found directly in SQL by

```
SELECT SNAME
FROM SUPPLIER
WHERE SNUM IS IN SELECT SNUM
                 FROM SP
                 GROUP BY SNUM
                    HAVING COUNT(PNUM) > 1
```

At present MURDER restricts the boolean in a HAVING clause to a single term, and the left part of this term must be a function reference which, as for a function reference in the select list, must evaluate a property of the group. The right part may be a literal (as above), another function reference, or a further nested sql_block.

The last line in Algorithm 7.1 above is changed to

```
if hav_ptr = null | satisfy(hav_ptr)
   do_group(group_ptr);
```

in which hav_ptr points to a pnode tree built from the boolean following HAVING. In the root of this tree, the

result and lson fields may point to a relnode and pnode respectively as described for a function in the select list. If the right side is also a function reference, a similar sub-tree is attached to the root's rson pointer, as in Figure 7.2 which represents the expression tree from

```
SELECT SNUM
FROM SPJ
GROUP BY SNUM
   HAVING COUNT(PNUM) = COUNT(JNUM)
```

which finds supplier numbers of suppliers who supply the same numbers of parts and jobs. (The restriction to one term in a HAVING condition is minor, and a simple modification should be sufficient to remove this arbitrary constraint.) As a first example of the last type of

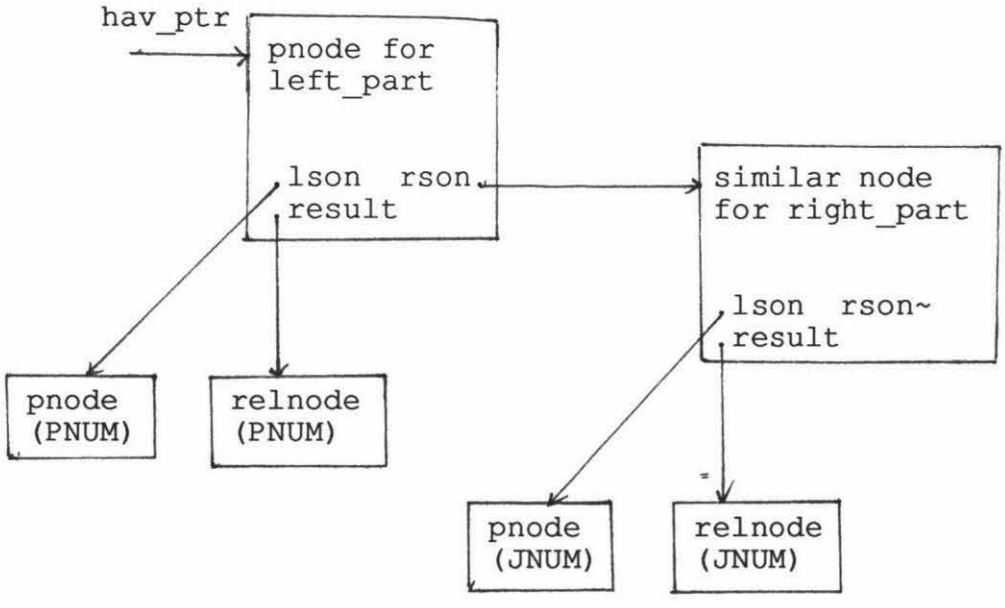


Figure 7.2 - Data Structures for HAVING fn θ fn

right_part, suppose it is required to find who supplies all jobs:

```
SELECT SNUM
FROM SPJ X
GROUP BY SNUM
   HAVING COUNT(JNUM) = SELECT COUNT(JNUM)
                        FROM SPJ
```

(The use of SPJ in the inner query also requires the aliasing of the SPJ instance in the outer query, even though no qualified reference uses it. This could have been avoided by making the inner query

```
SELECT COUNT(JNUM)
FROM JOB
```

but this would fail if any jobs are not currently represented in the SPJ relation. Integrity constraints require that there be a JOB-tuple including the JNUM-value from any SPJ-tuple, but the converse does not hold.)

In any case the nested query, being local, can be evaluated immediately. It should produce a scalar result, and this can be used in the same way as a simple literal.

To extend the query to find who supplies the same part to all jobs, all that is required is to change the GROUP clause to

```
GROUP BY SNUM, PNUM
```

in the above query.

This query illustrates a lack of definition which has existed in various forms in all versions since SEQUEL was

introduced. A sql_block is called a 'table expression', i.e. it defines a table (or relation) which in general may have any number of columns (attributes), and which may include any number of rows (tuples); yet in several places it is used in a more constrained way, as here when it is required to give a 'scalar' result, i.e. a table with exactly one row and one column. These constraints have already been met in Chapter 6.

Observe further that the nested query is only required to be scalar (in the sense described above); it need not be local, although of course its scope may not go beyond the scope of the enclosing GROUP-query; and it is now required to return a single value for each group. Suppliers of the same number of parts as jobs could be found by

```
SELECT SNAME
FROM SUPPLIER
WHERE SNUM IS IN SELECT SNUM
                 FROM SPJ X
                 GROUP BY SNUM
                 HAVING COUNT(PNUM) = SELECT COUNT(JNUM)
                                      FROM SPJ
                                      WHERE X.SNUM = SNUM
```

The innermost query could not refer to SUPPLIER's SNUM attribute (although its value would effectively be the same as X's) because this would violate the local requirement for the middle query. This raised a number of interesting questions, both in theory and for the implementation. Are two counts compatible (i.e. are they just integers, or are they related to the attribute being counted)? Is the innermost query <u>really</u> single-valued? (It is for a particular X.SNUM value.) This last is a basic

assumption in MURDER's solution of the implementation problem.

For the nested non-local query to be single-valued for each group, the qualification must be based on (the attributes in) the group list. This nested query may in turn involve a function, as in the example above, but the treatment is the same in both cases.

For an example of the case without a function, consider the relations

DEPT (DNUM, DNAME)

EMP (ENUM, DNUM, JOB, SAL)

and the requirement to list names of departments in which the average salary of employees exceeds the salary of the manager of that department. This can be expressed

```
SELECT DNAME
FROM DEPT
WHERE DNUM IS IN SELECT DNUM
                 FROM EMP X
                 WHERE JOB NOT = "MANAGER"
                 GROUP BY DNUM
                    HAVING AVG(SAL) > SELECT SAL
                                      FROM EMP
                                      WHERE X.DNUM = DNUM
                                        AND JOB = "MANAGER"
```

The local universe for the group query has already been evaluated when the right hand side of the having condition is met. (It can't be evaluated earlier, since it is not known whether the select list has to be temporarily expanded until the left hand function is determined; and later is too late, because there can be no interaction

between the ranges of any nested blocks in the WHERE and HAVING clauses.) The inner query contains non-local references, which means that a non-null pointer to the inner expression tree is returned from the sql_block call. A further call on the universe procedure at this point effectively expands the group's universe by one attribute (here SAL), and each tuple contains the value of this attribute (i.e. the manager's salary), which should be the same for each tuple in a group (although MURDER doesn't check this). The value of this attribute in the first tuple of each group can now be used as a pseudo-literal in determining whether the group satisfies the condition. (When a literal is involved, dptr is set to point to the stored value; here dptr is set to the appropriate offset from the first page of the current group relation, i.e. to this attribute in the first tuple in the group, since the same value occurs in each tuple.)

When the nested query contains a function, as in the earlier example, the local group_ptr references a tree for this function, and since this function is to be evaluated for each group, it should be attached to the enclosing hav_ptr's rson field. This hav_ptr is passed in as an additional parameter on the sql_block call, and, before returning, its rson field is updated if required. As in the earlier case, a further call on the universe procedure now ensures that any additional attributes required are present, and the grouping can proceed as before.

A further function is available in the HAVING boolean, the SET function. Its use can be illustrated by yet another solution to the problem of finding jobs which could be supplied entirely by supplier S2:

```
SELECT JNAME
FROM JOB
WHERE JNUM IS IN SELECT JNUM
                 FROM SPJ X
                 GROUP BY SNUM
                 HAVING SET(PNUM) CONTAINS SELECT PNUM
                                           FROM SP
                                           WHERE SNUM = "S2"
```

The innermost query returns a one-column table of PNUM-values supplied by supplier S2; the SET function extracts a similar table from a group. These two 'sets' may then be compared with one of the operators

```
[NOT] =
[NOT] CONTAINS
IS [NOT] CONTAINED IN
```

To implement these, a boolean function 'empty' is defined, with two parameters which are pointers to relnodes. Empty(A, B) is true if and only if A - B = ∅. It is not necessary to generate the difference, however, merely check until a tuple which would be in the difference is found, in which case the function returns false. Then

$A = B$ is equivalent to $n(A) = n(B)$ & empty(A, B)

$A \subseteq B$ $n(A) \leq n(B)$ & empty(A, B)

$A \supseteq B$ $n(A) \geq n(B)$ & empty(B, A)

with the other three equivalences following from the negation of one of these. Algorithm 7.2 shows function empty.

```
morea = first(A), moreb = first(B);
while morea & moreb
   if tA < tB return(FALSE);
   if tA = tB moreb = next(B);
   morea = next(A);
return(!morea);
```

Algorithm 7.2 - Empty

The SET function requires projection over the appropriate attribute(s), and the nested query (which must be local) is evaluated by a recursive call on the sql_stmt procedure, which returns a pointer to the result relation. A call to the empty function, passing in the pointer to the result of the projection and this result pointer, can now determine whether the current group is to participate in the result.

7.4. Binary Queries

For SQL this term applies to the set operators difference, intersection and union:

```
sql_block ::= local_expr setop local_expr

setop ::= INTER |
          MINUS |
          UNION
```

where local_expr is a SQL expression without external attribute references.

Thus, as in relational algebra, finding who does not supply part P1 by requires the difference

```
      SELECT SNUM
      FROM SUPPLIER
   MINUS
      SELECT SNUM
      FROM SPJ
      WHERE PNUM="P1"
```

(This indenting is used only to show the separation of two queries at the same level.)

Observe that if A and B are two union-compatible relations, their union cannot be written simply as

```
A UNION B
```

but must be expressed

```
      SELECT *
      FROM A
   UNION
      SELECT *
      FROM B
```

since set operators have expressions as operands, even though the table returned by

```
SELECT *
FROM A
```

is the same as A. Using the notion of inverse recursion might suggest

```
[A]
UNION
[B]
```

Implementation is straightforward. Although a binary query may be nested, since each operand is required to be local, by the time the parsing routines detect a setop token the (local) calculusnode list should consist of a

single node containing a pointer to the result of evaluating the expression at this level. A further call on sql_block will leave a similar situation before returning a null pointer. (If the pointer is not null the second operand contains an external reference.) The appropriate algebraic operation is then executed, attaching the result to the local calculusnode list head and discarding the following node.

Note that with this approach set operators are right-associative, i.e.

A - B - C is treated as A - (B - C)

7.5. Modifications to Calculus

Apart from the extensions already outlined in this chapter there are a number of relatively minor changes necessary from the implementation of the calculus described in Chapter 5, which are now summarized.

Since SQL encourages the user to write expressions involving 'local' queries which can be evaluated immediately, there is no attempt to optimize SQL expressions by pre-processing ranges. Except for queries expressed using the EXISTS or NOT EXISTS functions there is little to be gained; and anything written using these functions can be written without them in a manner which naturally leads to smaller derived relations and hence to improved processing efficiency. For this reason the procedure for evaluating the universe is split into two parts, the first of which

(called only by the calculus) carries out the range pre-processing before calling the second (called directly from SQL) for the actual evaluation. This approach means that the SQL routines do not have to build structures which are then simply discarded later on.

Because of the possibility that the select list will have to be evaluated once per group, this is left in the first of the two procedures, with appropriate calls included specifically from the SQL procedure.

The second procedure now requires a further parameter. For the calculus, this is the (global) head of the calculusnode list, as before; for SQL it is the local list head at the appropriate level of activation of sql_block.

In consequence of this a further change is required. From the calculus, the sequence of quantifier representations necessarily began with zeros corresponding to ranges from the target list, possibly followed by representations of quantifiers coupled to ranges in the qualification. In determining the size of the result, this list was examined, only updating the size when a universal quantifier was found; now however, since the 'local' target list may be associated with a quantifier, a means of distinguishing between ranges in the target list and in the qualifying expression was needed. This was achieved by accumulating, and passing in as an extra parameter, the count of attributes in local target list ranges.

The same problem arose in resolving universal quantifiers. Where previously it was sufficient to scan the list backwards until a value other than the representation of the existential quantifier was encountered, this condition was now augmented by a check that the (local) target list ranges had not been reached.

The SQL implementation here described runs faster than the equivalent calculus, confirming the remark made earlier that the tendency towards smaller ranges and intermediate results - consequent upon expression forms involving subqueries which can be evaluated locally - should lead to improved performance. The example query discussed in Chapters 4 and 5 required 24 page references for Algebra, 16 for Calculus (or 10 without range pre-processing), and requires 9 in the SQL implementation.

It should also be pointed out that SQL allows more direct expression of some types of queries, which leads to further improvements in efficiency. For example, the query about suppliers of all jobs: contrast

```
SELECT SNAME
FROM SUPPLIER
WHERE SNUM IS IN SELECT SNUM
                 FROM SPJ X
                 GROUP BY SNUM
                    HAVING SET(JNUM) = SELECT JNUM
                                       FROM SPJ

(s{SUPPLIER.SNAME): @j{JOB ]x{SPJ(s.SNUM=x.SNUM
                                & j.JNUM=x.JNUM)
```

Tests (calling the inner universe procedure directly from calculus) verify that, for small data volumes, bypassing

the pre-processing of ranges improves performance.

7.6. Updates

Section 3.3.2 has described the initial implementation of the MODIFY statement. In SQL, the syntax is

```
UPDATE rel_name [alias]
   SET assignment_list [ WHERE boolean ]
```

An alias may be required when another instance of the same relation occurs in the boolean. The assignment list is the same as before, viz.

```
assignment_list ::= assignment [ , assignment_list ]
assignment ::= att_name = expr
```

(As all updates merely change existing attribute values, it is never necessary to specify a new domain. At present any expression is only checked for reasonableness, and the syntax for expr does not allow nested queries, although they are permitted in SQL.)

Two differences are immediately apparent:

- the SET and WHERE clauses are in reverse order; and
- whereas for MODIFY the selection expression was based solely on the named relation, here the boolean, if specified, may be any of the standard SQL types.

The first difference required only minor modification to the existing routines, to allow an assignment to be followed by a where clause; the second involved rethinking

the whole process. Previously, the selection expression tree, if present, was used to identify tuples to be affected by the assignments. The existing SQL implementation was geared to producing a table of tuples which satisfy the boolean, and this might involve joining with other tables, in which case there is in fact no appropriate tree to modify. (For example, to change the supplier of part P7 to S6 could be done in two steps:

```
LIST SELECT *
     FROM SP
     WHERE PNUM = 'P7'
```

to indentify the current supplier, S1 say, and then

```
UPDATE SUPPLIER
SET SNUM = 'S6'
WHERE SNUM = 'S1'
```

Alternatively this could be achieved in a single step by

```
UPDATE SUPPLIER
SET SNUM = 'S6'
WHERE SNUM IS IN SELECT SNUM
                 FROM SP
                 WHERE PNUM = 'P7'
```

In this example, the existing routines return a tree which applies to the query universe, not just to the SUPPLIER relation, and the tree contains reference to attributes not belonging to SUPPLIER tuples, so that the initial approach could not be used.)

Instead, the approach to MODIFY was changed to that outlined in Algorithm 7.3. Here assptr points to a list of assignment trees, as for MAKE. Pkmod, fkmod and pkonly

```
if selection expression is present
   evaluate selp
else selp = null

while there is more input
   check_assignment(p, assptr, pkmod, fkmod, pkonly)

if selp
   create relation q like p
   sel(p, q, selp, null)
else q = null

modify(p, q, assptr, pkmod, fkmod, 1)
```

Algorithm 7.3 - Revised High-level Modify

are booleans indicating respectively whether the changes affect a primary key attribute, a foreign key attribute, and a primary key which is not part of a foreign key. In the last case, according to the rules given in Section 3.3.2, the selection expression must be based solely on primary key attributes. This is checked directly in the simple case, but for the SQL UPDATE it can only be checked indirectly, by insisting that the result of the selection contains exactly one tuple. Pkmod and fkmod are used in the modify routine of Algorithm 7.5, and the use of the last parameter is also discussed in that context.

The high level logic of the UPDATE statement is outlined in Algorithm 7.4. Thus, when procedure modify is called, either q is null - in the case when the modifications apply to all tuples - or q points to a relation containing only the tuples which are to be modified, and instead of

```
while there is more input
  and word != "WHERE"
    check_assignment(p, assptr, pkmod, fkmod, pkonly)

if there is more input
    selp = boolean()
    univ(calp, selp, ...)
    q = calp->relptr
else q = null

modify(p, q, assptr, pkmod, fkmod, 1)
```

Algorithm 7.4 - High-level Update

(as previously) applying the selection expression to each tuple, a check whether the tuple can be found in this relation determines if it is to be modified. This approach allows the universe procedure already implemented to be used. In Algorithm 7.5, the details of a number of points are glossed over. They are largely irrelevant here, but two deserve mention.

Function findtuple returns -1 if an attempt to add a new tuple fails because a reference cannot be satisfied, and 1 if the tuple is already present. In the former case this must be because a foreign key now references an unknown primary key. The first time this occurs, any tuples already in relation P are discarded, and all tuples from R are restored to P by reassigning the first page pointer. In the latter case, when tuples are merging, references from the duplicate tuple are removed, and the count of the survivor is set to the sum of the two counts. (This takes

```
R = makerel();
(copy P's attributes to R)
R->firstp = P->firstp, P->firstp = null;
R->numtup = P->numtup, P->numtup = 0;
t = calloc(1, P->relsize);
k = t + (P->head + n - 1)->aoffset;
moreR = first(R);
while moreR
   t = tR;
   if Q = null | moreQ & findtuple(Q, k, ' ')
      if fkmod
         (delete references from t)
      for (r = assptr; r; r = r->rson)
         evaluate(r, t);
      if findtuple(P, t, '+') < 0
         (error - back out and exit)
   else if findtuple(P, t, '+') > 0
      (tuples merge - adjust references)
   moreR = next(R)
                          /* now cascade changes */
if pkmod & P->numref > 0
   (prune assignment list)
   for (m = P->numref, f = P->reftab; m--; f++)
      (adjust pkmod, fkmod for f->refrel)
      modify(f->refrel, Q, assptr, pkmod, fkmod, f->ranum);
```

Algorithm 7.5 - Outline of Modify

account of consequent merging during recursive calls.)

The second point concerns these modifications at a lower level. For these referencing relations, the appropriate condition is that the foreign key from affected tuples is found in relation Q. Since all searches of real relations depend only on the primary key attributes, the last parameter to modify indicates the first attribute to be considered from candidate tuples. This parameter is 1 in the initial call, when the primary key is used, and is set to refer to the first attribute of the appropriate foreign

key when modifications are to cascade.

Although not as efficient as the former approach, this allows the full SQL boolean syntax to be available in update statements. All that remains to give a reasonably complete implementation (except for nulls) is the slight extension to allow a factor to be a nested query:

```
UPDATE R
SET a = SELECT MAX(b)
        FROM S
WHERE ...
```

Of course the nested query is implicitly constrained to return a scalar table, with exactly one row and column. Again, all the problems involved in this have already been addressed in other contexts.

CHAPTER 8

ANALYSIS AND CONCLUSIONS

This chapter summarises the work done in the three main areas, viz. domains, integrity (primary and foreign keys) and the SQL implementation.

8.1. Domains

A domain is a named, identifiable subset of some base data type, e.g. integer, character string, etc. Every candidate atomic value can be checked against the corresponding domain before it is accepted as part of a tuple. Such checks may include range or size checks, as well as checking type, and are known as domain integrity rules. (In addition, there are the relation integrity rules, which concern the admissibility of a tuple in a relation, or the relationship between tuples of two or more relations. These are discussed in the next section.)

The MURDER implementation includes only type checking, which is done when tuples are added either individually or by a high-level construct, when comparisons between attributes are required, and when expressions are involved. There is no range checking, and nor are there 'triggers' - indications that checking is appropriate at a nominated point in processing such as when a tuple is inserted or modified.

The absence of these checks is not a major problem in our environment. What has been implemented has been achieved

easily, and has significantly reduced the chance of nonsense queries being evaluated, with a consequent increase in the reliability of any results, since simple comparisons are now restricted to attributes with a common domain. It is possible to use any numeric values including literals in expressions, and to assign a domain to the result of such an expression with no more than 'reasonableness' checking. This however is no worse than the position in the majority of current programming languages, and indeed it is difficult to envisage a state when the possibility of misuse of data in this way is completely eradicated.

The simple implementation of domains as described is both useful and useable, and the cost of a fuller implementation would not necessarily produce an adequate return in real terms.

8.2. Integrity

Chapter 3 has described the creation and use of primary and secondary keys, and again this has been shown to impose no undue costs on operation. The returns in this case are considerable.

Entity integrity (ensuring the uniqueness of a primary key) places no further demands on processing, since MURDER's ordered physical storage requires a lookup whenever a tuple is to be inserted. Thus elimination of duplicates is not a costly procedure, as stated by many writers in the field, but merely part of normal processing.

Even without this ordering there are no extra costs. A relation has only one primary key, and hence only one place where checking for possible duplication is required - the primary index. (Recall that in MURDER the relation itself doubles as this.) Every insertion requires this to be updated, so that with an index but without ordering the same costs are incurred. (It is, however, true that MURDER's ordering allows more efficient elimination of duplicates during evaluation of queries than is the case in an unordered environment.)

Referential integrity, through foreign keys, has also been shown to be possible cheaply and efficiently, yet of more than 50 SQL products available, none supports entity and referential integrity, or domains, to any meaningful degree. (Date[87b])

It has already been pointed out that currently, if keys are supported, the only use which can be made of them is in this matter of integrity. Of course, keys <u>are</u> concerned with integrity, but that implies a wider use. In particular, although the existence of a foreign key only guarantees that a natural join with the referenced relation over the key attributes will not lose any information from the referencing relation, it also surely implies that such a join will be meaningful, and is likely to be required. Yet there is nothing to distinguish this from any other join. Suppose relation B carries a foreign key (with two attributes) to relation A, and that we require to join the two over this key. MURDER algebra allows

```
JOIN A
WITH B
WHERE A.key1 = B.key1
  AND A.key2 = B.key2
```

but this is an extension of the simple algebraic join, and the user might be required to

```
INTER[JOIN A
      WITH B
      WHERE A.key1 = B.key1]
 WITH[JOIN A
      WITH B
      WHERE A.key2 = B.key2]
```

The natural join is not appropriate here either, since this would make the intersection invalid. (It has already been argued that joining over a foreign key should be the natural join, with possible syntax

```
JOIN A TO B [ OVER att_name_list ]
```

where B has at least one foreign key to A, and the optional extension is only needed when there is more than one. The first attribute would be sufficient when - as in MURDER - a foreign key is required to consist of consecutive attributes; in general full enumeration would be necessary.)

A further problem which needs attention is that of keys - both primary and foreign - in the result of such an operation. (For simplicity the problem is discussed in terms of relational algebra.) Consider the relations SPJQ, SP and SUPPLIER, in which each of the first two includes a foreign key referencing the next. An attempt to 'join' all three, either of

```
JOIN [JOIN SPJQ
        TO SP]
  TO SUPPLIER

JOIN SPJQ
  TO [JOIN SP
        TO SUPPLIER]
```

requires the result of the inner join to have known keys, and the third relation either to reference or to be referenced by this result. Defining a real relation for the intermediate result would be adequate if two separate steps are acceptable, but this goes against the philosophy of the relational approach. Alternatively the definition of this join operation (an equijoin over a foreign key) could be extended, e.g.

> "The result of a 'key' join inherits the primary key from the first operand, together with references and foreign keys (except those between the relations joined) from both."

This would allow either of the above examples to produce the same result, i.e. associativity is maintained. (But what of keys for the result of other operations?) In the final approach, the syntax could be extended to include key-definition clauses, similar to those used in defining real relations.

There is no problem in calculus, as all joins are expressed in the same way, but in SQL this again raises questions. However, SQL has no specific terminology for the join, and the only place it could reasonably be introduced is in the from list, where it would frequently introduce unreferenced relations. An example of this is

given in the Section 8.3.3.

More serious is the accusation that, by introducing foreign keys, the relational system is reverting to the explicit inter-relationships of a CODASYL environment. In this case, an instance of record-type A 'owns' a subset of the instances of record-type B through a set, an explicit data structure declared in the schema and manipulated by its own commands. In such an environment, the SPJ database could be represented as in Figure 8.1, in which the connecting arrows now represent these sets. Any modification of the primary key of a SUPPLIER-record should not change the set of SP-records owned; this corresponds to the cascading of modifications through referencing relations in MURDER, since the association in a relational

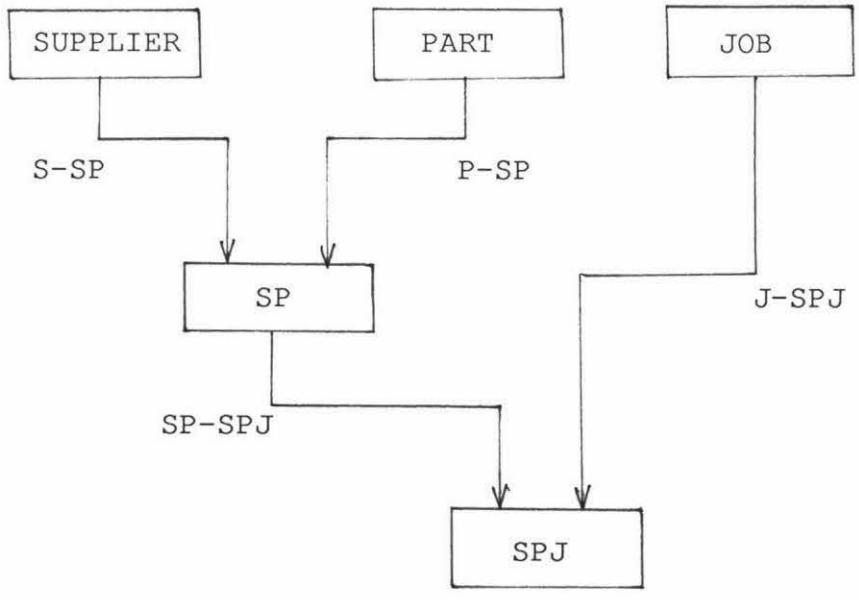


Figure 8.1 - Network Version of SPJ Database

environment is implicit, and from the 'bottom up'.

Similarly, modification of the primary key of an SP-record cascades to related SPJ-records, while the SP-record itself is 'reconnected', i.e. disconnected from the current S-SP and/or P-SP set instances and connected to new instances as appropriate; again, the MURDER approach mimics this. In fact it is difficult to justify any other approach.

Record deletion is a different problem. The simple network command

DELETE

will cause an error if the current record owns a non-empty set instance, just as MURDER refuses to delete a tuple currently referenced; this is the 'restrict' option of Date's classification, outlined in Chapter 2. (Network DBMSs often use ERASE instead of DELETE, but the latter is used here because it is familiar from the relational approach.) But network systems offer a variety of delete options, which depend on the membership conditions of the connecting sets. Set S-SP may have its membership declared to be either mandatory or optional. The former means that an SP-record may not exist in the database unless it is connected into an S-SP instance, while the latter means that it can.

DELETE MANDATORY

will have the effect of deleting owned SP-records when a

SUPPLIER-record is deleted in the mandatory case, with the operation being repeated at the next level. If however membership is optional, any owned SP-records are disconnected before the SUPPLIER-record is deleted - which corresponds to setting the foreign key null in a relational environment (supposing this is possible, i.e. nulls allowed, and the foreign key not overlapping the primary key of the referencing relation). This is Date's 'nullify' option, and is not possible in MURDER since null values are not supported. Two further options are

DELETE SELECTIVE

DELETE ALL

The former would also delete SP-records if all foreign keys would become null as a result of the current operation (which requires here that set P-SP also have optional membership), and the latter deletes all owned records regardless - the 'cascade' option for delete's.

It will be seen that to some extent the charge is proven, and that indeed implementation of referential integrity does compromise a basic relational claim regarding the implicit nature of data inter-relationships. Further, the user is now required to know a lot more about the likely effect of his operations, about the definition of the database, and about the semantic meaning of the data it contains.

8.3. SQL

In this section views on the SQL query interface are given under three headings:

- the suitability of calculus as a target for a SQL implementation.
- ommissions in the MURDER implementation - what has been omitted and why, and the significance of the omitted feature; and
- suggestions for improvements to SQL, which it is felt would make it less ambiguous or more usable.

There is no further discussion of SQL updates, which were not part of the original project, but which have in any case been fully dealt with when first described and in the section on integrity.

8.3.1. Suitability of Calculus

Since SQL is basically a calculus language, it is easy to map onto a calculus; but there are one or two additional points.

Although SQL is a calculus, it does not look like one. (This was in fact what Chamberlin et al set out to achieve.) Hence the mapping is not always obvious, although, as has been shown, it is always possible to achieve such a mapping in a well-defined way, which therefore makes the problem suitable for automatic processing.

Secondly, it has been shown that the nature of SQL makes it more efficient than a simpler calculus, which suggests that the direct mappings developed in this document are in some sense preferable to the (much) more complex mappings directly onto an algebra attempted by Ceri and Gottlob, whose published work to date only handles a limited subset of the range of possible queries (Ceri[85]).

It has also been shown that, while the authors of SEQUEL set out to eliminate quantifiers and the concepts of free and bound variables, they have succeeded only in hiding them from the user - these concepts are still present, and indeed necessarily so.

Finally, it is observed that the present tendency towards embedding relational processing within a host language such as COBOL or PL/1 is a consequence of the inherent 'granularity' of the relational approach - much processing is at the record (or tuple) level, while relational languages generally operate on relations, sets of tuples. (Once again Codd foresaw this problem, hence the 'piped' mode in Alpha - the FOR statement in MURDER adapts common programming terminology to meet the same needs.) The long-term solution, if indeed such a solution exists, lies in developing extensions of the relational language, so that this record-at-a-time processing is available. At present, and in the foreseeable future, a continued emphasis on embedding is necessary. Any solution not requiring embedding is likely to look very like current procedural languages since these reflect the inherent

nature of much processing.

8.3.2. Omissions in MURDER

Enough has already been said about domains and keys, so that further discussion of integrity is not warranted, and in any case SQL does not include these in any meaningful sense. Similarly no further discussion of the omission of explicit ordering of the result of a query will be given.

Currently MURDER does not support expressions in the select list, although as has been pointed out the basic mechanisms are present in the MAKE facility. This should be adapted, as it is a significant restriction on the usefulness of the expressions which the user can write, and indeed it will be adapted when time permits.

The restriction requiring attributes to precede functions in the select list of a group query is not serious. (Given that MURDER allows temporary relations to be created which can then be projected to give any required attribute ordering, the same effect can be achieved.) This restriction resulted directly from the basic approach taken of ordering all relations, which in this case meant that grouping could be easily achieved on tuple prefixes. A further reasonable restriction would be to require that if any attribute appears in the select list of a group query, all attributes in the group list appear. (This is reasonable because to omit any attribute is immediately to lose some of the information which has just been requested!)

(As an aside it may be mentioned that there is no intention to extend the calculus to include these features, although simple functions - as opposed to the image functions required for grouping - may be considered. In the context of the current courses, such extensions would not produce any significant advantage, and might result in a longer learning curve.)

Some SQL conditions have not been included, e.g.

WHERE expr BETWEEN expr AND expr

but these do not really impair the functionality of MURDER. Neither does the insistence on fixed ordering in conditions, i.e. restricting literals to the right part, etc., since this is the more natural mode of expression, even though symmetry in such things is usually permitted.

MURDER's insistence that a nested group query be 'local' also only expresses formally what would normally be the case.

At present the condition in a HAVING clause is restricted to a single factor. This has not produced any hardship in terms of not being able to write the solution to a query directly, but it is something which should be removed, and this is not seen as either difficult or time-consuming. All the relevant problems have already been addressed in other contexts. That the restriction exists at all can be traced back to the incremental nature of the implementation outlined below.

It is not seen as a restriction that the SET function is only available in the HAVING clause - that it should have been used in the WHERE clause in SEQUEL2 only came about because that version did not separate conditions applying to tuples and groups. In the first place, released SQLs do not all support SET, and in the second place it only makes sense here - its function is to extract a set of projected tuples from a group. What is more significant is the present inability to answer queries including conditions like

```
  ...
WHERE (SELECT ...
       FROM ...
       WHERE ...)
CONTAINS
      (SELECT ...
       FROM ...
       WHERE ...)
```

i.e. at present MURDER only allows set comparisons in conjunction with the SET function, and not with general queries which also return sets of tuples.

The time factor has been mentioned above. When the SQL interface was begun, the implications of some of the constructs made available in SQL were not fully understood, and hence the implementation proceeded along experimental lines, with new features being added one at a time. This incremental approach meant that some additions simply did not fit into the general approach taken elsewhere, because at the time they were not seen as part of this more general mechanism. Several of the features catalogued above will naturally be included when a reorganisation of the

SQL source code is undertaken.

8.3.3. Suggested Extensions

Some parts of SQL have been seen to be either confusing, or at best imperfectly stated, and there are some glaring cases when the implications of current theory have been ignored.

One example of the former is in the aliasing facility - in normal SQL the user need only specify an alias when the distinction between two instances of the same table must be made, while MURDER requires such aliases to be specified so that such a distinction is possible, even when the query does not require it. A much more satisfactory approach, in my opinion, would be to require (as in calculus) each table to have an alias - which may of course be the same as the table name - and then to require all column references to use the qualified form

```
alias . att_name
```

even though to some extent this would go against the original principal of not using the concepts of (free and bound) variables. For the user it is surely easier to do something always, rather than only sometimes, and then in circumstances which may not be fully understood. The syntax for specifying an alias is prone to error. MURDER algebra uses

```
rel_name = alias
```

and something like this would prevent WEHRE being treated as an alias in

```
 ...
FROM SPJQ
WEHRE ...
```

While discussing aliasing it is relevant to mention the problem of expressions in the select list. It should also be possible to give these a name, e.g.

```
SELECT att_name = expr ...
```

rather than use the default names 'EXPR1' ...; and in any case, if domains are supported, the result of an expression needs to have its domain specified, either explicitly, as

```
select_clause ::= SELECT * |
                  SELECT select_list

select_list ::= select_primary |
                select_list , select_primary

select_primary ::= alias . * |
                   alias . att_name |
                   att_name : domain_name = expr
```

or it could be inferred from the expression itself in simple cases. The notation here is suggested because it is consistent with CREATE, and with the MAKE statement described in Chapter 3. In a full realization of domains there would need to be some verification that the specified domain is appropriate for the result of the expression.

It is an unnecessary and irritating restriction of SQL that it should not be possible to replace

```
SELECT *
FROM rel_name
```

whenever it occurs as a complete expression, by just the relation name. SEQUEL, which did not use this form, would allow a query like

```
rel_name WHERE ...
```

In conjunction with the point about foreign keys made earlier, this could give something like

```
SELECT ...
FROM SUPPLIER KEYFROM SPJ
                      WHERE PNUM = "P1"
```

to find names of suppliers of part P1, although care would have to be taken to avoid introducing unreferenced relations into the from list. When, as will sometimes occur, the referencing relation carries more than one foreign key to the same referenced relation, a means of distinguishing between them would be required, for example that described in Section 8.2.

The reverse side of the same coin is the problem of recursion. It should be possible to replace the name of a relation by an (aliased) expression, since this yields a relation. Without this facility it is not possible to apply a function to the result of a function, so that even quite simple queries cannot be answered directly. Consider the query "Who supplies most of part P1?". (This

example can be answered directly in released versions of SQL; one could

```
SELECT SNUM, SUM(QTY)
FROM SPJQ
WHERE PNUM = "P1"
GROUP BY SNUM
ORDER BY SUM_QTY DESC
```

and only look at the first line in the result. This suggests that Alpha's quota facility might also be worth including:

```
SELECT (1) SNUM, SUM(QTY) ... )
```

A more direct way of achieving the same result would be to follow Date's suggestion that

```
SELECT COUNT(*)
FROM R
```

be replaced by the more 'natural' form

```
COUNT(SELECT *
      FROM R)
```

or even, adopting the convention above,

```
COUNT(R)
```

The former requires amending the syntax to

```
primary ::= ... |
            set_fn ([UNIQUE] fn_arg )

fn_arg ::= expr |
           table_spec
```

which is surely a natural way to include recursion, and would allow

```
SELECT SNUM
FROM SPJQ
WHERE PNUM = "P1"
GROUP BY SNUM
   HAVING SUM(QTY) = MAX(SELECT SUM(QTY)
                         FROM SPJQ
                         WHERE PNUM = "P1"
                         GROUP BY SNUM)
```

although this is still repetitive, and still imposes external implicit constraints on the 'shape' of the result of the nested query. Assuming that assignment to a temporary relation is possible we would rather

```
T := SELECT SNUM, TQTY = SUM(QTY)
     FROM SPJQ
     WHERE PNUM = "P1"
     GROUP BY SNUM
```

and then

```
SELECT SNUM
FROM T
WHERE TQTY = MAX(TQTY)
```

This would require another Alpha feature which is considered worthwhile - and which is available in MURDER -, the workspace, giving the user the ability to create temporary relations. But using MURDER's approach to recursion, and assuming that the user is familiar with the naming convention, these can be combined:

```
SELECT SNUM
FROM [SELECT SNUM, SUM(QTY)
      FROM SPJQ
      WHERE PNUM = "P1"
      GROUP BY SNUM] T
WHERE SUM_QTY = MAX(SUM_QTY)
```

Date[84] suggests that in the SELECT ... FROM ... format, the FROM part contributes nothing. There are two reasons

for disagreeing. Firstly, it does separate the variable declarations from their use, while retaining them in the same expression; to move this function elsewhere, while still providing an aliasing facility, could produce a very ponderous structure. Secondly, it allows the user to formulate queries which perhaps should not be allowed, although in fact Date uses one such query as an example. This second point could be removed by the very reasonable restriction that at least one attribute be referenced, in the select list, from each relation appearing in the from list. While on the same topic, there are grounds for a reordering of the clauses to

```
FROM from_list
SELECT select_list
 ...
```

which would have made the implementation so much easier, and would have allowed more consistent error recovery, without losing anything of the distinctive SQL flavour of constructing complex queries by composing simple formatted blocks.

Date[87b] comments on the 'redundancy' of SQL, i.e. its ability to express the same query in a number of (at least superficially) different forms. Consider the following solutions to finding names of suppliers of part P1:

```
(1)  SELECT SNAME
     FROM SUPPLIER, SP
     WHERE SUPPLIER.SNUM = SP.SNUM
       AND PNUM = "P1"
```

```
(2)  SELECT SNAME
     FROM SUPPLIER
     WHERE SNUM IS IN SELECT SNUM
                      FROM SP
                      WHERE PNUM = "P1"


(3)  SELECT SNAME
     FROM SUPPLIER
     WHERE SNUM = ANY SELECT SNUM
                      FROM SP
                      WHERE PNUM = "P1"


(4)  SELECT SNAME
     FROM SUPPLIER
     WHERE EXISTS SELECT *
                  FROM SP
                  WHERE SUPPLIER.SNUM = SNUM
                    AND PNUM = "P1"
```

(More solutions can be found!) Version (1) appears the simplest, but - as an attempt to negate the query shows - variable SUPPLIER is free in the boolean expression, but SP is bound to an (implicit) existential quantifier, i.e. while these concepts may be hidden they are nevertheless present. Version (4) makes these points explicit, although it would not have been allowed by SEQUEL2. Versions (2) and (3) show that both the 'IN' construct and the form '= ANY' are alternative ways of introducing this coupling in an explicit (athough still disguised) way. Probably this was the original motivation behind the IN form, but the implied equijoin approach proved too restrictive, requiring the introduction of the ANY modifier for queries like

```
WHERE att_name > ANY SELECT ...
```

Since NOT IN corresponds to NOT EXISTS, i.e. it introduces

the universal quantifier, the same reasoning could then have led to the ALL modifier, with its attendant problems. To find the part with maximum weight, a naive user might try

```
SELECT PNAME
FROM PART
WHERE WEIGHT > ANY SELECT WEIGHT
                   FROM PART
```

which actually gives all parts except those of minimum weight. In fact, the whole IN construct, which originally justified the use of 'Structured' in SEQUEL, could be dropped without loss of functionality, although it would reduce the expressiveness of the language.

The modifier 'ANY' could be dropped; it does nothing, except perhaps confuse, and should be left as the default, with 'ALL' being explicitly included if this is what is required. The query "Who supplies most of part P1" can be expressed

```
SELECT SNUM
FROM SPJQ
WHERE PNUM = "P1"
GROUP BY SNUM
   HAVING SUM(QTY) ≥ ALL SELECT SUM(QTY)
                         FROM SPJQ
                         WHERE PNUM = "P1"
                         GROUP BY SNUM
```

This is really back to using an explicit universal quantifier. If the temporary relation T has been created as earlier, the query could be expressed in calculus by

```
(t{T.SNUM): @u{T (t.TQTY >= u.TQTY)
```

To implement ALL requires only that the parameter in the nested sql_block call be changed; ANY is basically ignored in query processing. Such use of ALL is restricted to this type of query, and does not contradict the earlier assertion about the inability of SQL to apply a function to the result of a function, which results from the impossibility of using attributes from nested queries other than in WHERE or HAVING conditions. The recursive approach would move some such queries to the outer level, from which attributes may be retained.

The original designers of SQL deliberately left quantifiers out of their specifications. IBM's SQL/DS includes EXISTS and NOT EXISTS. If these are to be retained in the language (and I have shown that there is a place for them) then the IN query should be restricted to 'local' queries as previously described.

In general the syntax of a table_expr (in SEQUEL2 terminology) needs to be more precise, so that whenever a subquery is constrained by external conditions to produce only a certain number of rows and/or columns, this can be explicitly expressed, and checked. The present situation, where such constraints are only detected implicitly, is unsatisfactory. The MURDER implementation, which passes in a list of attributes to be matched, means that a breach of these constraints can be detected early, but the user may still be unaware of what is wrong, and there is nothing in the syntax to point out the error.

It is already argued that SQL is too hedged about with arbitrary restrictions, and this is true. This may have been caused by the designers' wish to make the language available to a wide range of users. More precise specification might in some way reduce this spectrum; but it could be productive, with the remaining users aware of what they really want, which can only lead to their being more easily able to get it.

It is questionable whether the full spectrum of users - from naive beginner to experienced professional - is possible, or even desirable. At present there are many pitfalls for the unwary, and in any case the need to embed much processing in a host program means that this processing is not directly available to some.

APPENDIX A

ALGORITHMS

The MURDER implementation on the VAX is written in the language C, and throughout this document program fragments are expressed in a C-like notation, with a number of simplifications which it is hoped will make the algorithms described more visible. The main simplifications are:

- instead of braces { and } to denote begin...end blocks, indentation is used;

- conditions are not written in parentheses;

- logical operators are written using single characters '&' and '|' instead of the doublets '&&' and '||' of C; similarly the predicate logical equality is written '=' instead of '==';

- string assignment, comparison etc. are written as in other common programming languages, instead of using procedures as is required in C.

Thus the C fragment

```
strcpy(str, "unknown");
if (c1 && strcmp(word, "relation") == 0)
{  strcpy(str + 8, word);
   if (more)
      getword();
}
else
   strcpy(str + 8, "attribute");
```

is given as

```
str = "unknown";
if c1 & word = "relation"
   str + 8 = word;
   if more
      getword;
else
   str + 8 = "attribute";
```

For those unfamiliar with C the last line requires comment. If str is defined as a character array, so that str[0] is the first character, a reference to str without subscript is treated as a pointer to str[0]; then str + 8 represents this pointer offset by (the size of) 8 (characters), i.e. a pointer to the 9th position in str.

In C '!' represents logical negation.

The for loop has the form

```
for (init; term; iter) statement
```

which is equivalent to

```
init;
while !term
   statement;
   iter;
```

The notation x++ is short for x = x + 1 with incrementing after reference, e.g. after the expression

```
str = "abcde", x = 0, ch = str[x++]
```

x and ch will have values 1 and 'a' respectively. Similarly, --x means decrement x before returning its value to the program.

The comma operator, illustrated above, separates expressions (an assignment is an expression) which are evaluated left-to-right, discarding all but the last - the value of this expression containing three assignments is 'a'.

The data declaration

```
type *p
```

specifies that p is 'pointer to type'; the '*' here is the dereferencing operator, so that '*p' other than in a declaration references the instance of 'type' addressed by p; and as in PL/1

```
p->field
```

references field in the type instance addressed by p.

APPENDIX B

SEQUEL Syntax

(from Chamberlin[74])

```
query ::= basic_query |
          basic_query ∩ query |
          basic_query ∪ query |
          basic_query - query |
          ( basic_query )

basic_query ::= [label :] sel_clause_list [where_clause]

sel_clause_list ::= sel_clause |
                    sel_clause_list , sel_clause

sel_clause ::= [SELECT expr_list FROM] table_name
                   [GROUP BY col_name_list] [dup_code]

dup_code ::= DUPL | UNIQUE

expr_list ::= expr |
              expr_list , expr

col_name_list ::= col_name |
                  col_name_list , col_name

where_clause ::= WHERE boolean

boolean ::= predicate |
            predicate AND boolean |
            predicate OR boolean |
            NOT boolean |
            ( boolean )

predicate ::= comparand comp_op comparand

comp_op ::= [ALL] rel_op [ALL] |
            set_op

rel_op ::= = | ≠ | < | ≤ | > | ≥

set_op ::= ⊂ | ⊆ | ⊃ | ⊇ | ⊄ | ⊅

comparand ::= expr |
              query

expr ::= atom |
         expr + atom |
         expr - atom |
         expr * atom |
         expr / atom |
         ( expr ) |
         literal
```

```
atom ::= col_name |
         table_name . col_name |
         label . col_name |
         constant |
         set_fn ( col_name )

set_fn ::= SUM | COUNT | AVG | MAX | MIN | SET

literal ::= lit_table |
            lit_tuple |
            constant

lit_table ::= { lit_tuple_list }

lit_tuple_list ::= lit_tuple |
                   lit_tuple_list , lit_tuple

lit_tuple ::= < constant_list >

constant_list ::= constant |
                  constant_list , constant

constant ::= 'string' |
             'number' |
             ø
```

APPENDIX C

SEQUEL2 Query Syntax

(from Chamberlin[76])

```
query ::= query_expr [ORDER BY ord_spec_list]

query_expr ::= query_block |
               query_expr set_op query_block |
               ( query_expr )

set_op ::= INTERSECT | UNION | MINUS

query_block ::= select_clause FROM from_list
                    [WHERE boolean]
                    [GROUP BY field_spec_list
                        [HAVING boolean]]

select_clause ::= SELECT [UNIQUE] sel_expr_list |
                  SELECT [UNIQUE] *

sel_expr_list ::= sel_expr |
                  sel_expr_list , sel_expr

sel_expr ::= expr |
             var_name . * |
             table_name . *

from_list ::= table_name [var_name] |
              from_list , table_name [var_name]

field_spec_list ::= field_spec |
                    field_spec_list , field_spec

ord_spec_list ::= field_spec [direction] |
                  ord_spec_list , field_spec [direction]

direction ::= ASC | DESC

boolean ::= boolean_term |
            boolean OR boolean_term

boolean_term ::= boolean_factor |
                 boolean_term AND boolean_factor

boolean_factor ::= [NOT] boolean_primary

boolean_primary ::= predicate |
                    ( boolean )
```

```
predicate ::= expr comparison expr |
              expr BETWEEN expr AND expr |
              expr comparison table_spec |
              < field_spec_list > = table_spec |
              < field_spec_list > [IS] [NOT]
                      IN table_spec |
              IF predicate THEN predicate |
              SET ( field_spec_list ) comparison table_spec |
              SET ( field_spec_list ) comparison
                      SET ( field_spec_list ) |
              table_spec comparison table_spec

table_spec ::= query_block |
               ( query_expr ) |
               literal

expr ::= arith_term |
         expr add_op arith_term

arith_term ::= arith_factor |
               arith_term mult_op arith_factor

arith_factor ::= [add_op] primary

primary ::= field_spec |
            set_fn ( [UNIQUE] expr )
            COUNT(*) |
            constant |
            expr

field_spec ::= field_name |
               table_name . field_name |
               var_name . field_name

comparison ::= comp_op |
               CONTAINS |
               DOES NOT CONTAIN |
               [IS] IN |
               [IS] NOT IN

comp_op ::= = | ^= | > | ≥ | < | ≤

add_op ::= + | -

mult_op ::= * | /

set_fn ::= AVG | MAX | MIN | SUM | COUNT | identifier

literal ::= ( lit_tuple_list ) |
            lit_tuple |
            ( entry_list ) |
            constant

lit_tuple_list ::= lit_tuple |
                   lit_tuple_list , lit_tuple

lit_tuple ::= < entry_list >
```

```
entry_list ::= entry |
               entry_list , entry

entry ::= [constant]

constant ::= ' string ' |
             number
```

APPENDIX D

MURDER Syntax

Session

```
session ::= [ stmt_seq ] Q[UIT]

stmt_seq ::= stmt [ stmt_seq ]

stmt ::= rel_name := alg_expr |
         rel_name := calc_expr |
         rel_name := sql_expr |
         utility_stmt
```

Algebra Expression

```
alg_expr ::= DIFF  alg_primary WITH  alg_primary |
             DIV   alg_primary BY    alg_primary |
             INTER alg_primary WITH  alg_primary |
             JOIN  alg_primary WITH  alg_primary join_spec |
             PROJ  alg_primary OVER  att_ref_list |
             SEL   alg_primary WHERE sel_expr |
             TIMES alg_primary BY    alg_primary |
             UNION alg_primary WITH  alg_primary

join_spec ::= OVER  att_ref [ , att_ref ] |
              WHERE join_expr

join_expr ::= join_term [ '|' join_expr ]

join_term ::= join_factor [ & join_term ]

join_factor ::= att_ref relop att_ref |
                ( join_expr )

sel_expr ::= sel_term [ '|' sel_expr ]

sel_term ::= sel_factor [ & sel_term ]

sel_factor ::= att_ref relop att_ref |
               att_ref relop sel_const |
               ( sel_expr )

sel_const ::= literal |
              function ( att_ref )

alg_primary ::= rel_name [ = alias ] |
                '[' alg_expr ']'
```

Calculus Expression

```
calc_expr ::= ( target_list ) [ : qual_expr ]

target_list ::= target_primary [ , target_list ]

target_primary ::= target_range [ . att_name ]

target_range ::= var_name [ '{' range_expr ]

qual_expr ::= quant_expr |
              simple_expr

quant_expr ::= quant_list ( simple_expr )

quant_list ::= quant_primary [ quant_list ]

quant_primary ::= quantifier var_name '{' range_expr

quantifier ::= [ ^ ] ']' |
               [ ^ ] @

range_expr ::= range_term [ '|' range_expr ]

range_term ::= range_factor [ & [ ^ ] range_term ]

range_factor ::= rel_name |
                 ( range_expr ) |
                 '[' calc_expr ']'

simple_expr ::= rel_expr [ => qual_expr ]

rel_expr ::= rel_term [ '|' rel_expr ]

rel_term ::= rel_factor [ & rel_term ]

rel_factor ::= calc_primary relop calc_primary |
               calc_primary relop literal |
               ( qual_expr ) |
               quant_expr

calc_primary ::= var_name . att_name

var_name ::= name
```

SQL Expression

```
sql_expr ::= SELECT select_list FROM from_list
                [ WHERE boolean ] [ group_clause ] |
             ( sql_expr ) |
             sql_expr setop sql_expr

update ::= UPDATE rel_name SET set_spec_list [WHERE boolean]

select_list ::= * |
                sel_ref_list [ fn_ref_list ] |
                fn_ref_list

sel_ref_list ::=  sel_ref [ , sel_ref_list ]

sel_ref ::= att_ref |
            qual . *

fn_ref_list ::= fn_primary [ , fn_ref_list ]

fn_primary ::= COUNT(*) |
               COUNT ( att_ref ) |
               function ( att_ref )

from_list ::= from_primary [ , from_list ]

from_primary ::= rel_name [ alias ] |
                 '[' sql_expr ']' alias

setop ::= INTER |
          MINUS |
          UNION

boolean ::= [ NOT ] boolean |
            boolean AND boolean |
            boolean OR boolean |
            IF boolean THEN boolean |
            att_ref_list IS [ NOT ] IN sql_expr |
            [ NOT ] EXISTS sql_expr |
            ( boolean ) |
            att_ref relop att_ref |
            att_ref relop literal

group_clause ::= GROUP BY att_ref_list HAVING hav_cond ]

hav_cond ::= hav_prim relop hav_prim |
             hav_prim relop sql_expr |
             hav_prim relop literal |
             SET ( att_ref_list ) [ NOT ] set_comp sql_expr

hav_prim ::= COUNT(*) |
             COUNT ( att_ref ) |
             function ( att_ref )

set_comp ::= = |
             CONTAINS |
             CONTAINED IN
```

Utility Statement

```
utility_stmt ::= CLOSE |
                 COUNT '[' argument ']' |
                 create |
                 DECLARE att_spec [ = value ] |
                 delete |
                 drop |
                 edit |
                 for |
                 FREE att_name |
                 function '[' argument ']' . att_name |
                 HOST database_name |
                 if |
                 insert |
                 LEAVE |
                 LET att_name = value |
                 LIST argument |
                 macro_name [ ( value_list ) ] |
                 make |
                 modify |
                 OPEN database_name |
                 report |
                 what |
                 WHERE domain_name |
                 literal

create ::= CREATE dom_spec |
           CREATE rel_spec

dom_spec ::= DOMAIN domain_name domain_type

domain_type ::= C length |
                D |
                I

length ::= unsigned_integer

rel_spec ::= RELATION rel_name ( att_spec_list )
                 PRIMARY KEY att_name_list
                 [ ; ref_list ]

att_spec_list ::= att_spec [ , att_spec_list ]

att_spec ::= att_name : domain_name

ref_list ::= ref [ ; ref_list ]

ref ::= FOREIGN KEY att_name_list REFERENCES rel_name

delete ::= DELETE rel_name < value_list > |
           DELETE rel_name [ argument ]

drop ::= DROP DOMAIN domain_name |
         DROP RELATION rel_list |
         DROP WORKSPACE [ rel_list ]
```

```
rel_list ::= rel_name [ , rel_list ]

edit ::= EDIT macro_name edit_seq Q[UIT]

edit_seq ::= edit_command [ edit_seq ]

edit_command ::= D range |
                 I lnum stmt_seq  \ |
                 L [ range ] |
                 P unsigned_integer |
                 R lnum stmt

range ::= lnum [ , lnum ]

for ::= FOR EACH rel_name . att_name stmt_seq ENDFOR

if ::= IF if_cond if_body ENDIF

if_cond ::= [ NOT ] EMPTY rel_name

if_body ::= stmt_seq [ ELSE stmt_seq ]

insert ::= INSERT rel_name < value_list > |
           INSERT rel_name argument

make ::= MAKE rel_name FROM argument
            SET result_spec_list

result_spec_list ::= result_spec [ , result_spec_list ]

result_spec ::= att_name [ : domain_name ] = make_expr

modify ::= MODIFY rel_name [ WHERE sel_expr ]
              SET set_spec_list

set_spec_list ::= set_spec [ , set_spec_list ]

set_spec ::= att_name = make_expr

make_expr ::= make_term [ addop make_expr ]

make_term ::= make_factor [ mulop make_term ]

make_factor ::= [ DB . ] att_name |
                literal |
                ( make_expr )

report ::= REPORT rel_name [ option ] report_group

option ::= PRINT |
           DISPLAY

report_group ::= heading_group report_group footing_group |
                 DETAIL line_group

heading_group ::= HEADING [ FINAL ] line_group
```

```
footing_group ::= FOOTING line_group

line_group ::= line_spec [ line_group ]

line_spec ::= LINE position field_list ;

position ::= [ + ] lnum

lnum ::= unsigned_integer

field_list ::= field_spec [ , field_list ]

field_spec ::= COL unsigned_integer field

field ::= literal |
          [ DB . ] att_name |
          SUM att_name

what ::= WHAT DB |
         WHAT DOMAIN |
         WHAT RELATION [ rel_name ] |
         WHAT WORKSPACE [ rel_name ]

argument ::= rel_name |
             alg_expr |
             calc_expr |
             sql_expr

value_list ::= value [ , value_list ]

value ::= string |
          literal

function ::= SUM |
             AVG |
             MAX |
             MIN

att_name_list ::= att_name [ , att_name_list ]

att_ref_list ::= att_ref [ , att_ref_list ]

att_ref ::= [ qual . ] att_name

qual ::= rel_name |
         alias

alias ::= name

att_name ::= name

database_name ::= name

domain_name ::= name

macro_name ::= name
```

```
rel_name ::= name

addop ::= + |
          -

mulop ::= * |
          /
relop ::= = |
          < |
          > |
          <= |
          >= |
          <>

literal ::= " string " |
            ' string ' |
            numeric_literal

numeric_literal ::= decimal |
                    integer

decimal ::= integer [ . unsigned_integer ]

integer ::= [ - ] unsigned_integer

unsigned_integer ::= digit [ unsigned_integer ]

name ::= letter [ alphanum_string ]

alphanum_string::= alphanum_character [ alphanum_string ]

string ::= character [ string ]

character ::= alphanum_character |
              special_character

alphanum_character ::= letter |
                       digit

letter ::= a | b | ... z | A | B | ... | Z

digit ::= 0 | 1 | ... | 9

special_character ::= any character except quote
                          not in letter or digit
```

BIBLIOGRAPHY

Bitton[83] D. Bitton et al, Benchmarking Database Systems: A Systematic Approach, Proc 9th International Conference on Very Large Data Bases (November 1983)

Ceri[85] S. Ceri & G. Gottlob, Translating SQL into Relational Algebra: Optimization, Semantics, and equivalence of SQL queries, IEEE Transactions on Software Engineering 11 No. 4 (April 1985)

Chamberlin[74] D.D. Chamberlin & R.F. Boyce, SEQUEL: A Structured English Query Language, Proc 1974 ACM SIGMOD Workshop on Data Definition, Access and Control, (May 1974)

Chamberlin[76] D.D. Chamberlin et al, SEQUEL2: A Unified Approach to Data Definition, Manipulation and Control, IBM Journal of Research and Development 20 No. 6 (November 1976)

Codd[70] E.F. Codd, A Relational Model of Data for Large Shared data Banks, CACM 13, No. 6 (June 1970)

Codd[71] E.F. Codd, A Data Sublanguage Founded on the Relational Calculus, Proc 1971 ACM SIGFIDET workshop on Data Description, Access and Control, San Diego, California (New York: ACM, 1971)

Codd[72] E.F. Codd, The Relational Completeness of Data Base Sublanguages, in "Data Base Systems" ed. Rustin, R (Engelwood Cliffs, New Jersey: Prentice-Hall, 1972)

Codd[85] E.F. Codd, Is Your DBMS Really Relational?, Computerworld (October 14, 1985); Does Your DBMS Run by the Rules?, Computerworld (October 21, 1985)

Date[82] C.J. Date, Null Values in Database Management, Proc 2nd British National Conference on Databases (Bristol, England: July 1982)

Date[83a] C.J. Date, Introduction to Database Systems, Vol II (Reading, Massacheusetts: Addison-Wesley, 1983)

Date[83b] C.J. Date, The Outer Join, Proc 2nd International Conference on Databases (Cambridge, England: John Wiley, 1983)

Date[84] C.J. Date, A Critique of the SQL Database Language, ACM SIGMOD Record 14, No. 3 (November 1984)

Date[86] C.J. Date, Introduction to Database Systems, Vol I 4th ed., (Reading, Massacheusetts: Addison-Wesley, 1986)

Date[87a] C.J. Date, A Guide to INGRES, (Reading, Massacheusetts: Addison-Wesley, 1987)

Date[87b] C.J. Date, Where SQL Falls Short, Datamation (May 1, 1987)

DBTG[71] CODASYL Data Base Task Group, APRIL 71 Report, (New York: ACM, 1971)

Engles[72] R.W. Engles, A Tutorial on Data Base Organization, Annual Review in Automatic Programming, Vol 7 Part 1 (Pergamon Press, 1972)

IBM[84] IBM Corp, SQL/DS Terminal User's Reference Guide, Document SH24-5017-3 (December, 1984)

McLeod[76] D.J. Mcleod, High Level Domain Definition in a Relational Database, Proc ACM SIGPLAN/SIGMOD Conference on Data: Abstraction, Definition and Structure (March 1976)

Palermo[74] F.P. Palermo, A Data Base Search Problem, Proc 4th International Symposium on Computer and Information Science, Maimi Beach (New York: Plenum Press, 1974)

Reisner[76] P. Reisner, Use of Psychological Experimentation as an Aid to Development of a Query Language, Research Report RJ 1707, IBM Research Laboratory (San Jose, California: January 1976)

Ullman[83] J.D. Ullman, Principles of Database Systems, 2nd ed. (Rockville, Maryland: Computer Science Press, 1983)

X3H2[83] X3H2 (American National Standards Database Committee) Draft Proposed Relational Database Language, Document X3H2-83-152 (August 1983)

X3H2[85] X3H2 (American National Standards Database Committee) Draft Proposed American National Standard Database Language SQL, Document X3H2-85-40 (February 1985)