AN APPROACH TO IMPLEMENTING A RELATIONAL DATABASE SYSTEM

H. G. Mackenzie

(September 1979)

I certify that, except where acknowledged in the text, the research contained in this report is entirely my own work.

[illegible]

26 November 1979

Hugh MacKenzie

CONTENTS

1.0 Introduction

Database management systems based on the CODASYL DBTG recommendations have become a de facto industry standard, and are available on the machines of most major manufacturers. These systems augment a higher level language such as COBOL, Fortran or PL/I with data manipulation commands, and hence using them requires programming an application in one of the host languages. This interface is at too low a level for the casual user. Many potential users are reluctant to make the initial heavy learning and programming investment required to use these systems effectively. In addition, the effort required after this initial investment, in programming each additional query, is considerable.

The situation would be vastly improved if the interface presented to the user was at a much higher level. The relational model, where the user views the data as a number of large tables, is one candidate for providing such a higher level interface.

In this paper, after giving a brief description of the relational model, I describe some currently implemented relational database management systems. Secondly, I describe the prototype implementation of a relational language, ALF, designed to act as a front end to a CODASYL database. ALF has been implemented as an interactive system at the CSIRO Canberra installation. It produces output in a language called CODE-A, and examples of the output may be found in Appendix E. This implementation involved setting up a mapping between the relational model and the CODASYL (network) model, and, from this mapping, deriving algorithms to translate commands in the relational language ALF into efficient programs suitable for execution on network databases.

The ALF translator may be regarded as a special purpose optimising compiler, as much attention has been paid to generating efficient code. The benefits of spending time on optimisation are even more clear cut with a database access language than with an ordinary programming language, as the programs being optimised are typically only a few lines long, and the extra time spent in translation can save many accesses to disc at execution time, in addition to saving central processor time.

As implemented at present, ALF does not contain any update commands, however their introduction would be straightforward, and the underlying algorithms would still be used if they were introduced.

The approach described in this paper has several novel features.

1) Many current relational database system implementations store all relation attributes, including foreign key attributes, explicitly in relation tuples. Join operations are performed by actual matching of the key values in the relations to be joined. ALF is implemented as a front-end to a CODASYL database system, and because of the use made of the

CODASYL set structure in implementing foreign keys, these keys are not explicitly stored. This would significantly increase retrieval efficiency, as well as introducing an important integrity constraint.

2) Implementing a relational interface to an existing database system allows the implementor to avoid most of the work which other relational system implementors have had to face, for example file and index structures, concurrent access, etc.

3) Although the network-relational correspondences have been pointed out or alluded to on previous occasions, for example, in (Nijssen 1974), (Olle 1975), (Sibley 1974), translation algorithms developed from them have not been previously published, to my knowledge.

4) The translation process generates an intermediate language which may either be interpreted directly or translated into any (reasonable) target language. Whether the command is to be interpreted or compiled, and what the target language is to be, is determined by the interchangeable final pass plugged onto the translator. This feature allows the possibility of having a common interface to different CODASYL Database Systems, perhaps running on different machines. The final pass currently included in ALF generates code in a language called CODE-A, which is described in Appendix C.

5) Unlike other relational systems, execution of a program generated by ALF does not involve the generation of any intermediate files. This contributes to increased retrieval efficiency.

The reader who simply wants some idea of the work described in this paper can read the sections 2 and 4, on "The Relational Model" and "Sample ALF queries". These two sections are self contained. The section on "Formal Definition of ALF" contains some material oriented towards understanding the sections which follow on the translation algorithms.

This project would not have been completed, or indeed started, without the encouragement and assistance of Dr J. L. Smith.

2.0 The Relational Model

There have been a great many articles published on Relational Database systems, thus only a brief and incomplete survey will be presented here. In particular, the important topics of functional dependency and normalisation will not be covered.

The relational model of data was first proposed in (Codd 1970), a paper which, together with (Codd 1971a,1971b) produced a flood of research into Relational theory and practice. The Relational model used in this paper is the one described in Codd's early work; recent extensions, for example those described in (Codd 1979), are not included.

Basically, the relational model describes a way in which a database user views the data in a database. There is no requirement that this view shall correspond to the way in which the data is stored. The data may be considered as being arranged in a number of tables or matrices, each one called a *relation* and given a name. Each table or relation contains a number of named columns, where the data in each column is of a type selected from some homogeneous underlying domain. The name of each column is called an *attribute* or *attribute name* of the relation. Each row of the relation is called an *n-tuple* or simply *tuple*. An example of a relation representing Departments is shown in Figure 2.1.

Department

D #	Dloc	Dname	
123	ACT	CSIRO	
124	ACT	Treasury	
125	ACT	Health	
126	ACT	Defence	
127	Melbourne	Telecom	

Fig. 2.1

One central assumption of relational theory is that in each relation, there is a subset of the attributes of the relation whose values uniquely identify the relation tuple in which they occur; that is, by appealing to the semantics of the system being modelled by the relational database system, it is known that two tuples may not have the same values of these identifying attributes. This constraint is assumed to be enforced by those programs which maintain the database. In addition, this subset is minimal, that is by removing an attribute from the subset, one destroys this uniqueness property. Such a subset of the attributes of a relation is called a *candidate key*. A candidate key which never has any of its component attributes undefined is selected and called the *primary key*.

For a particular application, the totality of relations and attributes can be considered to model the application. This model is called a *relational schema.*

A relation in a relational database may represent a class of objects (in the widest sense) in the real system being modelled by the database system. Two objects in the system may be related in some way. One way that relational database systems represent this is to have the identifying attributes (that is, the primary key or one of the other candidate-keys) of one relation occurring in another. When this occurs the attributes in the second relation are known as a *foreign-key.* The foreign key would not in general be a candidate key of this second relation. Two relations may also be related using non candidate key attributes.

As an example, assume a relational database contains two relations representing companies and departments, where each of the departments belongs to one of the companies. This situation could be presented in the way shown in Figure 2.2.

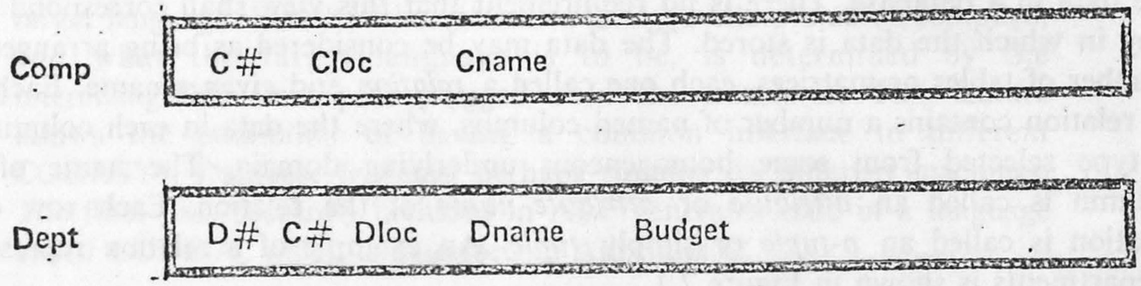


Fig. 2.2

If C# is the primary key of the Company relation, the situation where a particular department belongs to a particular company could be represented by the C# attribute of the tuple representing that particular department in the Department relation being equal to the C# attribute in the appropriate tuple of the Company relation. In this case, C# is a foreign key in the Department relation. C# could also be part of the primary key of Department (for example, if the primary key was [C#,D#]), but need not be (for example, if the primary key of Department was [D#]). It should be emphasised that, in representing some relationship between two objects in this way, there is no requirement for the foreign key attributes to have the same names as the primary key attributes, although this is very often the case.

3. Other Relational Database System Implementations

This section first gives a classification of the types of language that have been proposed for manipulating data stored using the Relational Model, and secondly fits the methods used in the ALF implementation into context by describing some other Relational Database System Implementations. There have been many such implementations, and this report will mention only three of them, viz PRTV, System-R, and INGRES. The selection has been based on the fact that the systems described do not implement the operators specified in the relational language in a brute force, straightforward way, but make significant transformations and optimisations in translating from the operations at the logical level of the Relational language to the physical level at which the data is actually stored.

Languages for accessing data stored using the Relational Model have been classified in (Chamberlin 1976) under the following headings :

Relational Calculus Oriented Languages

Relational Algebra Oriented Languages

Mapping Oriented Languages

Graphics Oriented Languages

Natural English query languages could be added to this classification.

In Calculus oriented languages each relation may be thought of as a predicate in a first order predicate calculus, and each tuple may be thought of as a ground instance of such a predicate. A statement in a calculus oriented language contains a *qualification* which selects a subset of the tuples in the database. A *targetlist* selects attributes from the retrieved tuples, and a *command* operates on the selected values, outputting them or performing some other computation. The *qualification* is a formula in a first order predicate calculus, and may contain universal and existential quantifiers in some languages (Codd 1971b).

In Algebra Oriented languages a number of unary or binary operators are defined on relations, and produce new relations. These operators include Projection, Restriction (Filter, or Selection), Join, Division and the set operators Union, Intersection and Difference. There is an assignment operator, used to assign the result of a relational algebraic expression to an intermediate result relation, which may, in turn, be used in other expressions. Relational Algebra has been discussed in detail elsewhere, for example in (Codd 1972b), and will not be treated here.

Algebra and Calculus based languages are equivalent in the sense that that an expression in a Calculus based language may be transformed into a statement in

Relational Algebra. An algorithm for this transformation is given in (Codd 1972b). The algorithm was developed with the aim of demonstrating the equivalence, independent of any implementation. The efficiency questions raised by this algorithm were addressed in (Palermo 1972).

Mapping Oriented languages are languages such as SEQUEL (Chamberlin 1974). They comprise nested *mappings* ; a *mapping* being a block of code which maps a known attribute or set of attributes into a desired attribute or set of attributes. The result of one such mapping may be used in specifying another mapping.

Se
g

Graphics Oriented languages, of which Query-By-Example (QBE), (Zloof 1975,1977), is the best known, operate by having the user fill in blank spaces in a predefined form, or blank relation. It is claimed in (Thomas 1975), that this approach facilitates learning to use a relational language. This assertion is less obviously true for more complex operations than for the simpler ones. In other ways graphics oriented, or tabular languages appear to be equivalent to relational calculus based languages, and translatable to them in a straightforward manner.

3.1 PRTV

PRTV, (Peterlee Relational Test Vehicle) is described in the series of papers by Hall and Todd, as well as in (Verhofstad 1976) and (Owlett 1976). It is a system developed at the IBM UK Scientific Centre at Peterlee, and has been used for some large applications. Its user interface, ISBL, (Information System Base Language), is based on Relational Algebra.

The underlying relational database files, called *bricks*, are stored sorted by leading attributes, common leading attributes being suppressed, and other attributes being compressed. Text values are stored in an area separate from the relation tuples themselves.

The algebraic operators union, intersection, difference, select, join and project are implemented at the ISBL level, and are specified in infix notation. There is an assignment operator. The ISBL expression is transformed to a language called *CIL*, (Common Intermediate Language). In this form the expression is called a *cilstring*, and is essentially the ISBL expression tree in a linearised, prefix notation.

The ISBL user may supply a number of assignment statements, any one of which may use the result of a previous statement. Relation names may be used as variable identifiers, or new variable identifiers may be introduced as a result of an assignment statement. Each time an identifier is used, it is bound either by value or by name. If value binding is used, the current relation value is inserted into the expression. This is presumably done by copying the whole relation. If name binding is used, the relation name is inserted, and the relation tuples are materialised at the time the expression is evaluated. Name binding enables any changes to the database to be reflected in the answers to queries, and hence is used to define different *views* of the data. There is another, less frequently used binding type, called *binding by expression*, which is described in (Owlett 1976).

The Algebraic operations in each statement are not carried out at the time that the statement is input, but are deferred until one of the following occurs.

a) The user lists a result relation.

b) The user asks the cardinality of a result relation.

c) The user requests that the result relation be explicitly materialised, and stored as a brick.

d) The user converts the result relation to a *relational file* . Relational files allow a users program to access the relation as a sequential file, one tuple at a time.

When one of these operations occurs, the expression tree is optimised, and evaluated to produce the result tuples. The latest published status of the optimisation stage is given in (Verhofstad 1976). Verhofstad distinguishes two types of optimisation; global and local. Global optimisation deals with issues of database organisation, such as what indexes to maintain, and tuple placement control. These issues are discussed in (Hall 1975a).

Local optimisation is furthur divided into algebraic optimisations, which use relational algebraic identities to transform the query tree into an equivalent one, and non-algebraic optimisations, which transform the tree using such performance improving measures as file inversion. Local optimisation is discussed in (Hall 1975) and (Verhofstad 1976).

Local optimisation performs the following sorts of transformations on the query tree.

* Filters (that is, selectors) are moved as far down the tree as possible. This causes them to be executed as early as possible, reducing the sizes of the relations that have to be handled.

* Multiple adjacent projections are merged into one projection.

* Projections which remove leading attributes, on which relations are sorted, require that the result be resorted, and are moved towards the leaves of the tree for earliest possible execution. This reduces the amount of data in each tuple that must be handled.

* Common subexpressions are identified, transforming the tree into a lattice. Each common subexpression need only be evaluated once,

transformed to a brick, and reaccessed when necessary.

* The most efficient implementation of the relational operators, particularly join, is estimated in a particular case. Indexes are used where possible.

* Idempotency laws for relational and boolean algebra are applied to simplify the expression.

* Various more complicated tree transformations, particularly involving the use of indexes, are applied. (see Verhofstad 1976)

Tree transformations similar to those used in PRTV are also discussed in (Smith 1975), in reference to the Relational Algebraic system SQUIRAL.

Se 9.

After the tree is transformed, a process is associated with each relational operator, or internal node. Full materialisation, or *realisation* of intermediate files is avoided as much as possible. Each process on an internal node makes calls to the processes on the children of that node to materialise a single result tuple. This tree of processes is similar to the List Set Generator method used in (Mackenzie 1977c). Realisation is necessary for some projections, where a file must be sorted to remove duplicates, and resorted using leading attributes as sortkeys. A node where a full realisation is necessary is called a *break point* .

3.2 System-R

System-R is one of the better known and most well developed Relational Database Systems. Its user interface language is SEQUEL (Chamberlin 1974). System-R consists of a Relational Storage System (RSS), whose Interface language is the Relational Storage Interface (RSI). The RSS is concerned with managing devices, space allocation and paging, locking, deadlock detection and backout, recovery, and with maintaining images and links, which are described later in this section.

On top of the RSS, and interfacing to it is a Relational Data System (RDS), which is accessed via an interface called the Relational Data Interface (RDI). RDI provides facilities closely parallel to those in SEQUEL, although it will also support other systems such as QBE (Zloof 1975,1977). A programming language is interfaced to the RDS using a *cursor*, which identifies a set of tuples called the *active set* of the cursor. One can associate a SEQUEL statement with a cursor, and retrieve tuples satisfying the statement into locations in the user program using a FETCH call.

The RDS contains an optimiser which chooses an algorithm to satisfy the query from the access methods supported by the RSS. It is this optimiser which is of most interest in this report.

Relation tuples are stored in *segments*; and one segment may contain tuples from a single relation or from more than one relation.

Each tuple in a relation is identified by a *tuple identifier* , or TID. A TID corresponds closely to a *Database-key* of CODASYL, being an efficient, hardware-address oriented pointer to individual tuples.

The RSS makes explicit use of two structures, *images* and *links*.

An *image* is a B-tree index structure containing non-truncated keys. It provides fast access on a single attribute of a whole relation, and as the whole key is maintained in the index, can be used in retrieval without accessing the tuples themselves.

Leaf pages of an image are linked together in a doubly linked list.

Images may be *clustering*, in which case the tuples are maintained physically sorted according to the image key, or *nonclustering*. It follows that there may be only one clustering image for each relation.

An image resembles a SORTED, INDEXED, PRIOR PROCESSABLE set with OWNER SYSTEM, in CODASYL terminology. A *clustering* image corresponds to the case where the record (relation) on which the image is defined has LOCATION MODE VIA the SET corresponding to the image.

A *link* is a mechanism for connecting tuples, and may be *unary* or *binary*. A *unary* link is a logical ordering on a single relation, and resembles a SORTED SET with OWNER SYSTEM, in CODASYL terminology. A *binary* link connects a single tuple in one relation with all the tuples in another relation, such that the values of a particular attribute in the first, or parent, tuple equal the values of a particular attribute in the second, or child, set of tuples. A tuple participating in a link is joined, using TID pointers, to its prior and next twins in the link. Tuples must be inserted into a link individually at the RSS level. A link therefore resembles a MANUAL , non information bearing SET, where the SET membership is defined on the equality of certain data items in the owner and member records. Both types of link are PRIOR PROCESSABLE, in CODASYL terminology, as link members are connected with next and prior pointers.

Links and images may be created or destroyed at any time. The pointers which implement images and links are TIDs and are stored as an affix to the tuple data. This affix may be expanded and contracted as images and links are created and destroyed.

The RDS may provide the RSS with clustering hints based on value ordering or on grouping associated tuples in a binary link.

In summary, the storage structures used in System-R closely resemble a subset of those available in CODASYL systems, with the difference that in most CODASYL systems these structures cannot be created and destroyed.

The optimiser in the RDS begins by classifying the SEQUEL statement into one of several classes. The first class contains statements operating on a single relation, the second contains those containing a join term, and a third contains those which, in addition, use the GROUP BY option. Secondly, the optimiser examines the system tables to find images and links which could assist in executing the statement. Thirdly, a set of reasonable methods for executing the statement is derived, and lastly, cost estimates for each method are computed, and the method with minimum cost executed to produce a result.

For each relation, the system tables contain the following information.

R: The Relation cardinality.

D: The Number of data pages the relation occupies.

T: The average number of tuples per page (R/D)

For each image, I, the image cardinality, (The number of distinct field values in the image), is maintained.

A coefficient H, the number of tuple comparisons equivalent to one page access, is estimated and stored.

Consider the case where there is a single relation query with a predicate containing a conjunctive term of the form *(attribute) (relational-operator) (value)*. A number of cases arise. There may be no image, a clustering image, or a nonclustering image defined for the attribute. The relational operator may be "=" or not. The relation may occupy a file by itself, or there may be other relations in the file as well. To execute the query, either the whole relation may be scanned, or the image may be used. The properties of the predicate, together with the relation properties maintained in the system tables, are used to estimate the cost for one of eight methods for executing this type of query and to select one of them.

Consider a two relation query whose predicate contains a join term and a restriction on each relation in the query. There are a number of possible methods for evaluating such a query, using clustered or unclustered images on one or both relations, binary links between the relations, and whole relation scanning, possibly sorting the relations. A method is chosen which depends on the access paths actually available, and on whether the parts of the predicate involving one relation only are expected to be highly selective or not.

When a method for executing a query is chosen, it is compiled into an *optimised package*, or OP. This OP is bound to the cursor defined in the program making calls to the RDS, and a result tuple produced incrementally whenever a FETCH is done using that cursor. This avoids the generation of intermediate files.

3.3 INGRES

INGRES (Integrated Graphics and Retrieval System) was developed by Stonebraker and others at the University of California, Berkeley. It runs under the UNIX operating system on PDP 11 machines (model 34 or higher). The system is described in (Stonebraker 1976).

The user interface is via an interactive language, QUEL, which is calculus based, and allows aggregate functions to appear in the qualification. There is a version of QUEL callable from a higher level language. This version, called EQUEL, or Embedded QUEL, allows *piped mode* or tuple at a time retrieval into variables in a users program.

INGRES has an underlying storage structure which is paged, and in which each tuple has a TID similar to the TID in System-R, and similar to the CODASYL database-key. Five file structures are used. They are sequential or *heap*, *hashed*, *compressed hashed*, *ISAM* and *compressed ISAM*. Secondary indexes may be specified for any file. Each relation is stored as one such file, and there are no explicit links between files, except for those implied by the equality of attributes in different relations.

The hashed access methods provide retrieval given an exact value for the key attribute; ISAM in addition provides retrieval over a range of key item values.

The access methods all have a common interface, so that details of the access method's implementation is hidden from the higher level query execution processes. Adding a new access method is therefore straightforward, provided that it conforms to the existing interface conventions.

QUEL supports both retrieval and update, however only retrieval shall be explicitly considered here.

The query optimisation algorithms in INGRES operate by *decomposing* a query into a number of single variable queries. These queries are executed using a process called the OVQP (One Variable Query Processor). The reduced ranges are used in evaluating the residue of the query using *tuple substitution*. This process is similar to the process of pushing projections and restrictions back through joins in PRTV or in SQUIRAL (Smith 1975).

The OVQP uses the file access method or an available index to materialise a set of tuples satisfying a one variable query (from a single relation), to project the tuples onto that subset of the attributes needed for later processing, and to format them as a file indexed on a key to be used later in processing the residual query.

After having evaluated those single variable queries that can be detatched, the results of the evaluations are used to substitute attribute values in the residual query, creating a series of simpler queries. This process is equivalent to materialising the cartesian product of the reduced ranges of the relations processed by the OVQP, and evaluating the residue of the qualification.

A more sophisticated decomposition algorithm is given in (Wong 1976), which gives an algorithm to be implemented in a later version of INGRES.

Se
9.

4.0 Sample ALF statements

In this section I will show the capabilities of the calculus based relational language ALF which is the main topic of this paper. This will be done here using a set of example retrieval statements designed to demonstrate the language; A more formal definition will follow in Section 5. A larger set of example queries which were translated using the ALF translator, together with the output produced, is given in Appendix E. Syntactically the features of ALF are similar to, and to some extent modelled on, those of QUEL, although the underlying implementation, storage structures, and execution strategy is totally different.

4.1 Examples

The relational schema to be used is illustrated in Figure 4.1.

Comp	C# Cloc Budget
Dept	C# D# Dloc Budget
Emp	C# D# E# Name Address Age
Project	Proj# Finish-Date Complete
Projdept	Proj# C# D# Liason-person
Comprel	Parent# Sub# Numshs

Fig. 4.1

The system being modelled by this schema contains a set of Companies, represented by the COMP relation which has primary key C#. Each Company has a number of Departments, represented by the DEPT relation, with primary key (C#, D#) and each Department has Employees, represented by EMP with primary key (C#, D#, E#).

There are Projects, represented by PROJECT with primary key PROJ#. Each Department may be associated with a number of Projects, and each Project with a number of Departments. The association of a particular Project with a particular Department is represented by a tuple of the PROJDEPT relation. PROJDEPT contains two foreign keys, (D#, C#) indicating the Department participating in the association, and PROJ# indicating the Project. In addition there is a SUPPLIER relation with primary key S#, a PART relation with primary key P#, and a SUPPLY relation. The SUPPLY relation contains S# and P# as foreign keys. Each tuple in the SUPPLY relation represents the fact that Supplier S# supplies Part P#.

Each of these sample retrieval commands will consist of a command, following by a *targetlist* of attribute values to be retrieved and operated on by the command, and a logical condition following the word "WHERE". This logical condition, or *qualification*, restricts the set of values returned in the target list.

Se
9

4.1.1 Retrieval using one relation only.

Give the name and address of all employees who are over 60.

```
OUTPUT EMP.NAME, EMP.ADDRESS WHERE EMP.AGE GT 60
.
```

In this query, the command is OUTPUT, the targetlist is EMP.NAME, EMP.ADDRESS, and the qualification is EMP.AGE GT 60. The attribute names NAME, ADDRESS and AGE are qualified in this query by EMP, which is the relation that they come from.

In general however, attribute names in the target list are qualified by a *variable*, or *relation variable*, which may be thought of as ranging over all the tuples of a particular relation. Conceptually, the variable takes each tuple of the relation in turn as its value. In ALF, relation names do double duty as relation variables referring as variables to the corresponding relation. When a relation variable which is not a relation name is used, it must be declared in a RANGE statement before being used. Thus an equivalent way to express the previous query is

```
RANGE OF X IS EMP.
OUTPUT X.NAME, X.ADDRESS WHERE X.AGE GT 60.
```

The relation variables take all the tuples in their range as value, and for each combination of tuples, the qualification is evaluated. If the qualification is true, the targetlist is accepted. The targetlist is in some respects like a virtual relation, to be operated on by the statement command, except that there is not necessarily a primary key, and duplicate tuples are not removed.

4.1.2 Retrieval using a join between two relations.

Give the name and address of employees who work for companies located in the ACT.

```
OUTPUT EMP.NAME, EMP.ADDRESS
WHERE COMP.CLOC EQ "ACT"
AND COMP.C# EQ EMP.C# .
```

In this case the qualification reads "..the company's location is "ACT" and the company's company number is the same as the employee's company number". The latter conjunct in the qualification is called a *join term*. Recall that the presence of the foreign key C# in EMP indicates the company to which each particular EMP tuple belongs (C# being the primary key of COMP).

4.1.3 Use of disjunction

Give the name and address of employees who work for companies which are either located in the ACT or which have budgets greater than ten million dollars.

```
OUTPUT EMP.NAME, EMP.ADDRESS
WHERE COMP.C# EQ EMP.C#
AND [COMP.CLOC EQ "ACT" OR COMP.BUDGET GT 10000000].
```

4.1.4 Use of expressions in qualification

Give the department names of departments whose budgets are more than 20% of their companies budgets.

```
OUTPUT DEPT.NAME
WHERE DEPT.BUDGET GT 0.2 * COMP.BUDGET AND
DEPT.C# EQ COMP.C# .
```

4.1.5 Multiple joins

Give the name and address of liaison people whose projects are not complete after the first of May, 1979, and who work for company XYZ, department ABC.

```
OUTPUT EMP.NAME, EMP.ADDRESS WHERE
EMP.C# EQ "XYZ" AND EMP.D# EQ "ABC"
AND PROJECT.FINISH-DATE LT 790501
AND PROJECT.COMPLETE EQ "NO"
AND PROJECT.PROJ# EQ PROJDEPT.PROJ#
AND PROJDEPT.LIASON-PERSON EQ EMP.E#
AND PROJDEPT.D# EQ EMP.D#
AND PROJDEPT.C# EQ EMP.C#.
```

4.1.6 Introduction of aggregate functions in qualification

Find all departments whose budgets are greater than the average departmental budget (that is, the average for all companies).

```
RANGE OF D1, D2 IS DEPT.
OUTPUT D1.D# WHERE
D1.BUDGET GT AVG(D2.BUDGET).
```

Se
9

4.1.7 Another example using an aggregate function

Find all departments whose budgets are greater than the average departmental budget for their own companies.

```
RANGE OF D1,D2 IS DEPT.
OUTPUT D1.D# WHERE
D1.BUDGET GT AVG(D2.BUDGET WHERE D2.C# EQ D1.C#).
```

In example 4.1.6, D2 ranges over all tuples in the DEPT relation, and computes the average budget. D1 ranges over all tuples of the DEPT relation a second time, accepting those tuples which satisfy the qualification, that is whose budgets are greater than the previously computed average.

In example 4.1.7, D1 ranges over all the tuples of the DEPT relation, and for each tuple, the average is computed by D2 ranging over all the DEPT tuples which have the same C# as the D1 tuple. There is scope for optimisation here, but this optimisation is not the concern of the ALF user.

It is important to note that the text which follows the aggregate function AVG, is merely another query in the form

targetlist WHERE *qualification*

This query is evaluated, and the aggregate function applied to the collection of tuples which result from the evaluation. The number of items in the targetlist must equal the number of arguments expected by the aggregate function. Aggregate functions currently available in ALF are MEAN, AVG, MAX, MIN, RANGE, COUNT, TOTAL, SSQ, EXISTS, ALL.

These last two functions provide facilities equivalent to existential and universal quantification in the query qualification, and greatly extend the power of ALF retrieval statements. For a fuller description of these see Section 5.3, and the examples in Appendix E.

4.1.8 Use of two aggregates in retrieval statement

Find companies which have average departmental budgets less than 30000 or a maximum departmental budget greater than 100000.

```
RANGE OF D1,D2 IS DEPT.
OUTPUT COMP.C# WHERE
MAX(D1.BUDGET WHERE D1.C# EQ COMP.C#) GT 10000000
OR AVG(D2.BUDGET WHERE D2.C# EQ COMP.C#) LT 30000.
```

In this example, the variable COMP ranges over the tuples of the COMP relation, and for each tuple, D1 and D2 individually range over all the DEPT tuples to compute the MAX and AVG aggregate functions. There is scope here for optimisation of the execution of this query, but again, this is not the concern of the ALF user.

4.1.9 The Homogeneous Hierarchy example.

Augment the database schema of Figure 4.1 with the relation COMPREL, shown in Figure 4.2.

Comprel	Parent# Sub# Numshs

Fig. 4.2

As before, each tuple of the COMP relation represents a company. Each tuple in the COMPREL relation represents the fact that the company with Company Number PARENT# has a subsidiary with Company Number SUB#, and that the parent holds NUMSHS shares in the subsidiary.

Example a)
Find all the subsidiaries of company "XYZ", and the number of shares company "XYZ" holds in each.

```
OUTPUT COMPREL.SUB#, COMPREL.NUMSHS
WHERE COMPREL.PARENT#="XYZ".
```

Example b)

Find the names and locations of subsidiaries of companies located in the ACT where the subsidiary's budget is greater than ¾ of the parent's budget,

```
RANGE OF SUB,PARENT IS COMP.
OUTPUT SUB.CNAME,SUB.CLOC WHERE
PARENT.CLOC EQ "ACT" AND
SUB.BUDGET GT 0.75 * PARENT.BUDGET AND
COMPREL.PARENT# = PARENT.C# AND
COMPREL.SUB# = SUB.C# .
```

This process can be continued for as many levels as needed, however it does illustrate a deficiency of ALF as currently implemented. The number of levels is always fixed in the query, that is, there is no transitive closure operation as described in (Zloof 1976). This means that there is no mechanism for issuing a query such as "Find all the subsidiaries of company XYZ, and all their subsidiaries, and so on..."

Se
9

4.1.10 Existential Quantification

Output companies which have at least one department located in the ACT.

```
OUTPUT COMP.C# WHERE
EXISTS(DEPT.D# WHERE DEPT.DLOC="ACT" AND
DEPT.C#=COMP.C#) IS TRUE.
```

4.1.11 Universal Quantification

Output companies, all of whose departments are in the ACT.

```
OUTPUT COMP.C# WHERE
ALL(DEPT.D# WHERE DEPT.C#=COMP.C# IMPLIES
DEPT.DLOC="ACT") IS TRUE.
```

For all department tuples in the DEPT relation, if the department belongs to the company being tested, it must be in the ACT. If it does not, the implication is trivially satisfied, as the antecedent is false.

The EXISTS and ALL functions are really predicates over one or more relations, rather than functions over attributes selected from relations. Only the variables in the targetlist are significant, not the attributes. EXISTS is true if there is a combination of the targetlist variables which satisfies the qualification. ALL is true if the qualification is satisfied for all possible targetlist variable combinations.

5.0 Formal Specification of ALF

The ALF language as currently implemented was designed to assist in the development of the algorithms which translate it into operations on a CODASYL database. As it stands it contains a statement for retrieval only. To be a generally useful stand alone language it would have to be augmented with update commands, with extra options on the retrieval command (for example, to perform report generation and sorting) and with an interactive capability. The introduction of any of these would not invalidate the translation algorithms which are the subject of this paper.

Nor would these algorithms be invalidated by the choice of a different input language; although ALF is based on the relational calculus, it would be possible for a language based on relational algebra or a language such as Query by Example to serve as an input to this translation process.

5.1 ALF Syntax

ALF is parsed by recursive descent. The grammar is shown in the following set of syntax diagrams. There is one diagram for each nonterminal symbol in the grammar. The name of each nonterminal symbol appears on the top left of each syntax diagram, and legal constructs in the language are constructed by following the flow lines in the diagram from left to right. The flow lines may loop back to indicate repetition; this is indicated by appropriately pointing arrows. Terminal or nonterminal symbols may appear in the diagram.

An ALF retrieval statement consists of one of the commands OUTPUT or PRINT followed by a query, which consists of a targetlist followed by a query, which in turn consists of a targetlist followed by a qualification clause. Using the previously described conventions, the syntax of the first part of retrieval statement is illustrated in the following way

```
retrieval statement

      +- OUTPUT -+
   --+            +-----  query -----  .
      +- PRINT  --+
```

The full syntax follows

```
statement

            +---- range statement ----+
            |                         |
  ------+                              +----------------
            |
            +-- retrieval statement --+
range statement

  -- RANGE -- OF -- variable list -- IS - relation list

retrieval statement

          +- OUTPUT  +
      ---+           +------- query ------.
          +- PRINT --+

query

          -targetlist-qualification clause ------

targetlist

                    +-------- , ------+
                    |                 |
                    V                 |
                         expression

qualification clause

                    WHERE ------ boolex ------

boolex

                +--- IMPLIES -----+
                |                 |
                V                 |
           ---------- implicand ----------

implicand

                +------ OR -------+
                |                 |
                V                 |
           ---------- disjunct -----------

disjunct

                +-- NOT --+
                |         |
                +---------+----- AND ------+
                |                          |
                V                          |
           ----------- conjunct ----------

conjunct

                +---- [ boolex ] ---------+
                |                         |
           --------------------- term -----------
```

Se
9

```
term

                         +------- EQ ------+
                         |                 |
                         +------- NE ------+
                         |                 |
                         +------- GT ------+
                         |                 |
                         +------- GE ------+
                         |                 |
    expression ------------------ LT ------------expression --
                         |                 |
                         +------- LE ------+
                         |                 |
                         +------- IS ------+
                         |                 |
                         +----- EQUALS ----+
                         |                 |
                         +-------  =  -----+
                         |                 |
                         +-------  <  -----+
                         |                 |
                         +-------  >  -----+

expression

                    +---- + ----+
                    |           |
                    +---- - ----+
                    |           |
                    V           |
                ------- aterm -------

aterm

                    +---- * ----+
                    |           |
                    +---- / ----+
                    |           |
                    V           |
          ----------- afactor -----------

afactor

                    +---- ** ----+
                    |            |
                    V            |
          ----------- aprimary --------------

aprimary

     ----------------           item         -------
            |                                   |
            +---- arith function  ( arglist ) --+
            |                                   |
            +---- subquery ---------------------+
            |                                   |
            +---- numeric constant -------------+
            |                                   |
            +---- logical constant -------------+
            |                                   |
            +---- string  constant -------------+
            |                                   |
            +---- ( --- expression ) -----------+
```

subquery

```
        --- aggregate-function - ( -- query -- ) ---
```

agregate-function

```
              ------ AVG -------------
                 +-  MAX  -+
                 +-  MIN  -+
                 +-  MEAN -+
                 +- TOTAL -+
                 +- RANGE -+
                 +-  SUM  -+
                 +-  SSQ  -+
                 +-EXISTS--+
                 +-  ALL  -+
                 +-  SUM  -+
                 +-  SSQ  -+
                 +-   SD  -+
```

item

```
              -relation variable . attribute name -
```

arglist

```
                       +------ , ------+
                       |               |
                       V               |
               --------- expression ------------
```

arith-function

```
              -----+-  SIN  -+-----
                   +-  COS  -+
                   +-  TAN  -+
                   +-   :   -+
```

Informally, the target list consists of expressions built up from *items*, which are dotted pairs consisting of a relation variable and an attribute name. The relation variable must either be a relation name or must have been previously declared in a RANGE statement. The relation referred to by a particular relation variable is called the *range* of that variable. Each relation variable may be thought of as taking a tuple from it's relation as it's value. The attribute name must be an attribute name in the relation referred to by the relation variable. The item pair selects the attribute value from the relation tuple which is the current variable value.

The qualification consists of a logical expression built up by using the logical operators AND, OR, AND NOT and IMPLIES. The precedence implied by the syntax diagrams may be altered by use of brackets in the usual way.

Terms consist of expressions connected by relational operators. Terms of the form R1.D1 *relational operator* R2.D2 are called join terms; and those join terms where the relational operator is EQ play a special role in that they are (usually) used to connect one relation with another, and may allow one of the relations to be accessed from the other using a CODASYL set.

Expressions are composed of items of the form *relation variable . attribute name*, linked in the usual way using arithmetic operators and arithmetic functions. The variable in an item is said to qualify the attribute. The value of an item is the value of the named attribute in the relation tuple which is the current value of the variable.

Each term may be thought of as a function of the relation variables in it, taking the values true or false.

5.2 Evaluation

The evaluation of an ALF command may be visualised in a number of ways. The following way, taken from (Codd 1972b) but omitting the steps concerned with universal and existential quantification, will be used in this paper.

1) Take the *cartesian product* of the ranges of all the relational variables which occur in the query. If two or more variables range over the same relation, then that relation will occur in a cartesian product with itself in the final product. The cartesian product, sometimes called *full quadratic join*, is as defined in (Codd 1972b).

2) Reject those tuples in the cartesian product for which the qualification is false. If the qualification contained subqueries it would be necessary to invoke this evaluation process recursively to evaluate the query contained in the subquery, and hence to evaluate the qualification.

3) Project the result of 2) onto those items which occur in the target list. This projection does not eliminate duplicate tuples, but removes those items not specified in the target list.

4) Compute any expressions in the target list for each remaining tuple.

5.3 Aggregate Functions

In general, a command may contain Aggregate Functions, which operate over the tuples retrieved by the associated query. Duplicate tuples are not removed before evaluation of the aggregate function, unlike some other relational languages.

The Aggregate Functions currently included in ALF are given below. Additional functions, including user-defined functions, could be added without difficulty.

1) MEAN, AVG
Compute the mean of the values of the targetlist expression.

2) MAX, MIN
Compute the maximum or minimum of the values of the expression in the targetlist.

3) RANGE
Compute the difference between the maximum and minimum values for the expression in the targetlist.

4) TOTAL, COUNT
Compute the total number of tuples returned as a result of the following query.

5) SUM
Compute the sum of the values of the targetlist expression.

6) SSQ
Compute the sum of the squares of values of the targetlist expression.

7) SD
Compute the standard deviation of the values of the targetlist expression.

8) EXISTS
This function returns the boolean constant TRUE if there are any tuples satisfying the qualification of the query governed by the EXISTS function. If no tuples satisfy the qualification, FALSE is returned.

9) ALL
The ALL function has a single argument. Let the argument be the item RV.DI, and let the range of RV be R. The function returns TRUE if all the tuples in the R relation satisfy the qualification, otherwise it returns FALSE. For example, the following would be TRUE if all employees in the database were under 50 years old.

ALL(EMP.E# WHERE EMP.AGE LT 50)

5.4 Nesting of queries

The qualification of the subquery may itself contain other subqueries, and so on, although more than two levels would be unusual. This leads to a hierarchy of queries in a statement, which may be represented graphically as in Figure 5.1.

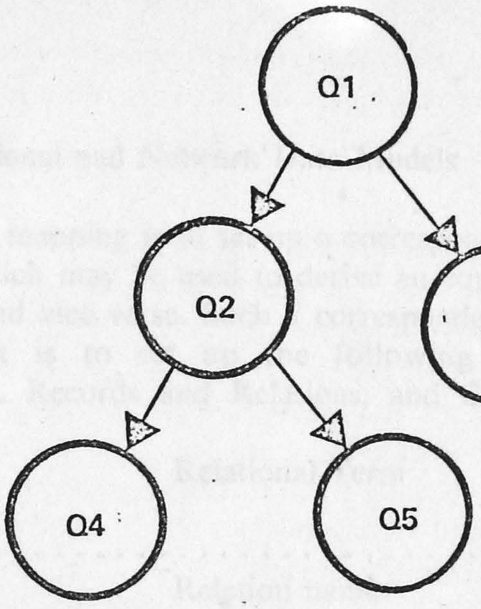


Fig. 5.1

This graph is called the *Q-graph* for the ALF statement. The query at the topmost level of the Q-graph is called the root query of the command. Each other Q_i represents a query in a subquery, that is, a query operated on by an aggregate function.

Variables which occur in a query may also occur in subordinate queries, that is queries further towards the leaves of the Q-graph. These variables are called *global variables* in the subordinate query, and, as assigning values to those global variables assigns a value to the subquery, the subquery may be thought of as a function of the global variables.

Variables occurring in a query which are not propagated from a higher query in the Q-graph are said to be *local* to the subquery. Thus example 4.1.7 contains a query Q1 and a subquery Q2, and has the structure

```
OUTPUT Q1
where Q1 is D1.D# WHERE D1.BUDGET GT SQ2(D1.C#)
and in which SQ2 has the structure AVG(Q2)
and where Q2 has the structure
D2.BUDGET WHERE D2.C# EQ D1.C#
```

D1 is global to Q2 and local to Q1, and D2 is local to Q2.

The subquery may also be thought of as a function of those items occurring in the targetlist or qualification which have any of the global variables of the subquery as the relation variable.

A variable may only be local to a single query. It would be possible to introduce an Algol-like name scoping rule so that the same name used in queries at the same level in the Q-graph would refer to a different variable. This is not done in ALF. If a name is used in this way, a generated name is used instead, and an informative diagnostic issued.

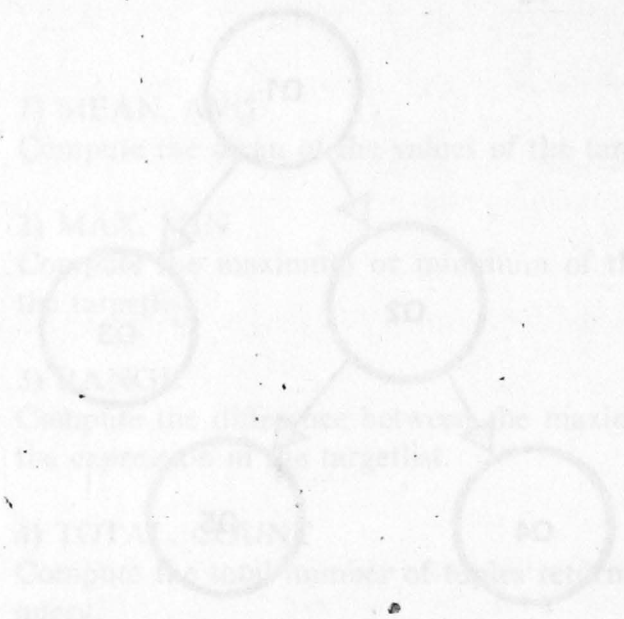


Fig. 6.1

This graph is called the *Q-graph* for the ALF statement. The query at the topmost level of the Q-graph is called the root query of the command. Each other Q represents a query in a subquery, that is, a query operated on by an aggregate function.

Variables which occur in a query may also occur in subordinate queries, that is queries further towards the leaves of the Q-graph. These variables are called global [illegible] in the subordinate query [illegible] variables assigns a value to the subquery, the subquery may be thought of as a function of the global variables.

Variables occurring in a query which are not propagated from a higher query in the Q-graph are said to be local to the query. This example [illegible] contains a query Q1 and a subquery Q2 and has the structure:

```
        OUTPUT Q1
[illegible] (D1.D[illegible] WHERE [illegible] SET OF SQ(D1.C[illegible]))
      and in which Q2 has the structure AVG(Q2)
      and where Q2 has the structure
      (D2.SUBJECT WHERE D2.C[illegible] EQ D1.C[illegible])
```

D1 is global to Q2 and local to Q1, and D2 is local to Q2.

The subquery may also be thought of as a function of those items occurring in the targetlist or qualification which have any of the global variables of the subquery as the relation variable.

A variable may only be local to a single query. It would be possible to introduce an Algol-like name scoping rule so that the same name used in queries at the same level in the Q-graph would refer to a different variable. This is not done in ALF. If a name is used in this way, a generated name is used instead, and an informative diagnostic issued.

6.0 The Mapping Between the Relational and Network Data Models

The first step in defining this mapping is to set up a correspondence between a relational and a network schema which may be used to derive an equivalent network schema from a relational schema, and vice versa. Such a correspondence is suggested in (Olle 1975). A starting point is to set up the following fairly obvious correspondence between CODASYL Records and Relations, and CODASYL Data Items and Relation Attributes.

Network Term	Relational Term
Record Type	Relation name
Record Instance	tuple of relation
Data item in record	Attribute in relation

Before defining the correspondence, note that the word *coset* is used throughout the rest of this paper to mean *set* in the CODASYL sense, following (Nijssen 1975).

A relational schema may be derived from a network schema by applying the following two steps.

1) Starting at the top of each hierarchy (at each record which is not a member of a non SYSTEM owned coset), propagate the primary key of each record down through the coset making it a data item of the member record of the coset. If data items propagated in this way are actually stored in the member record, the coset is said to be *non information bearing* (Metaxides 1975). If the data items are not stored in the record, but are implicitly defined by the coset, the coset is said to be *information bearing*, and the data items are called *virtual data items*. This propagation may be continued down a hierarchy of cosets if necessary, virtual attributes being used as source attributes. This is done for each hierarchical path in the network schema.

2) Define a relation corresponding to each record type, and define attributes of the relation corresponding to the data items of the record. *Virtual data items* in the network schema become attributes in the relational schema. They are called *virtual attributes*. The user at the relational level need not be aware that the virtual attributes are not actually stored in the database.

In this paper, all cosets are information bearing. If a coset is non information bearing in the original network schema, that is, data items exist in the owner and member records whose equality defines the coset occurrence in the schema, then those data items must become virtual items in the member record.

The correspondence between a coset in a network schema and the equality between a primary key and a foreign key in a relational schema is the lynchpin of the whole translation process used in ALF.

The reverse transformation from a relational schema to a network schema may be carried out by identifying foreign keys in relations and defining a coset between that record as member, and the record containing the foreign key as its primary key as owner. Foreign key names need not be the same as the names of the corresponding primary keys, although, conventionally, they often are. Foreign keys must be chosen with the semantics of the underlying data in mind.

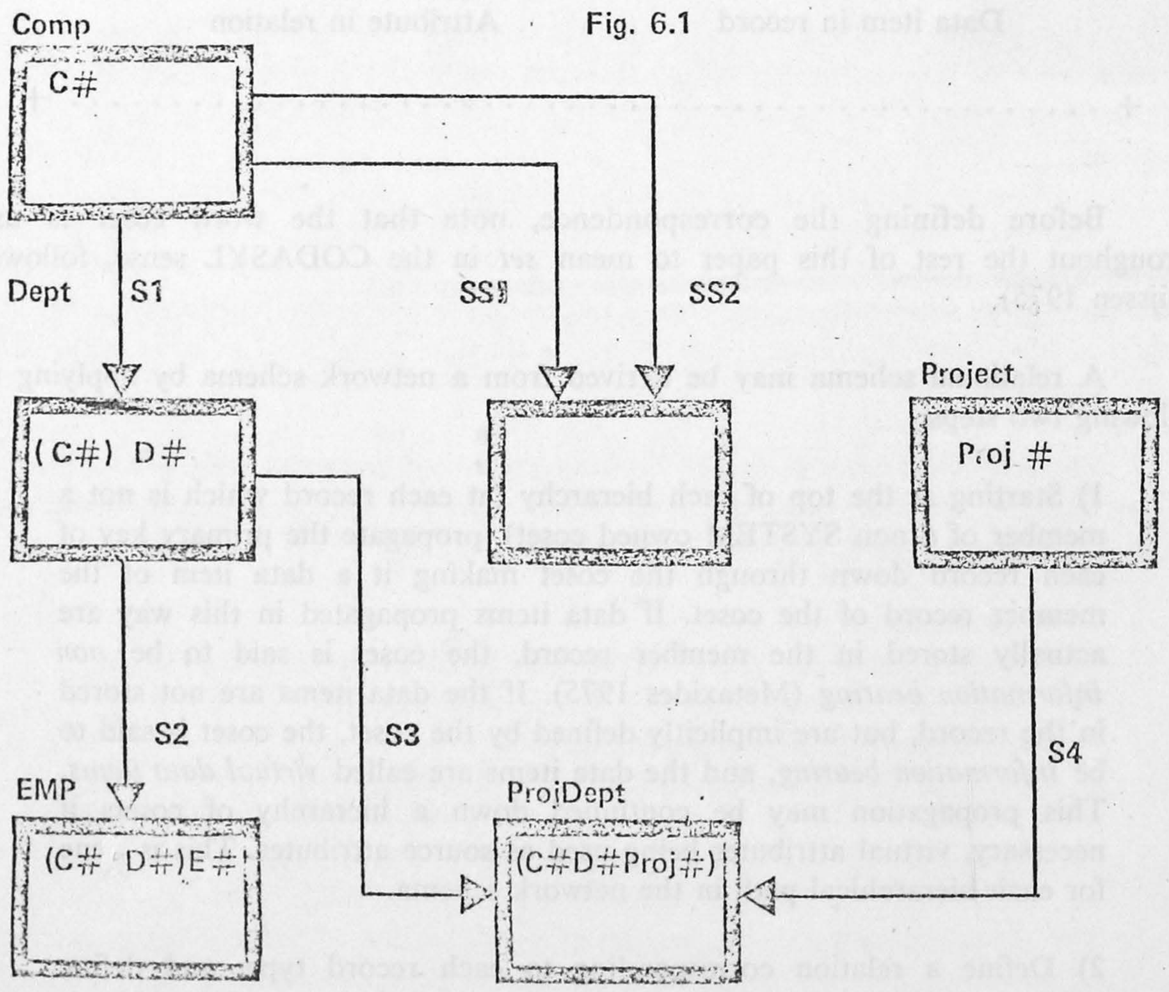


Fig. 6.1

Each chosen foreign key becomes a virtual data item in the member record of the derived coset.

A graphical notation introduced in (Bachman 1969) is often used to represent the coset interconnections in a network schema. This notation, called a Bachman

Diagram, is called the C-graph of the network schema in this paper. The C-graph for the database used in the Examples in Section 4 is given in Figure 6.1, given that those data items with the same names as primary keys of other records were chosen as the foreign keys.

The network entity which corresponds most closely to the relational variable in ALF is a variable which takes a database key as its value. A database key does in fact identify a network record occurrence, corresponding to a relation tuple, at least inside a single run-unit. The database key variable must be constrained to refer to one type of record only.

Thus, three extra correspondences are added to the table given earlier in this section

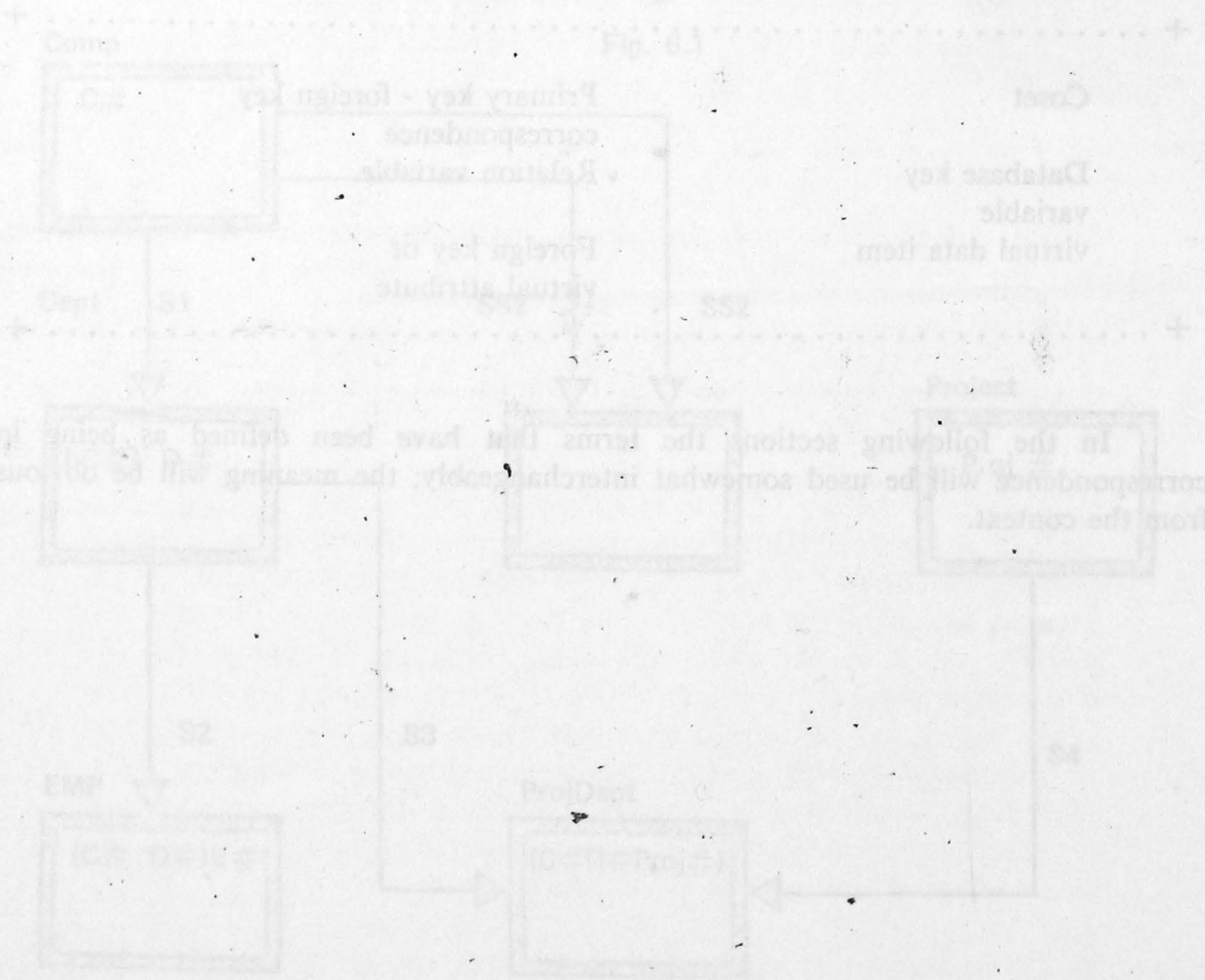
Network Term	Relational Term
Coset	Primary key - foreign key correspondence
Database key variable	Relation variable.
virtual data item	Foreign key or virtual attribute

In the following sections the terms that have been defined as being in correspondence will be used somewhat interchangeably; the meaning will be obvious from the context.

data items must become virtual items in the member record.

The reverse transformation from a relational schema to a network schema can be carried out by identifying foreign keys in relations and defining a coset between [illegible] key as owner. Foreign key names need not be the same as the names of the corresponding primary keys, although, conventionally, they often are. Foreign keys must be chosen with the semantics of the model in mind.

Fig. 6.1



Each chosen foreign key becomes a virtual data item in the member record of the derived coset.

A graphical notation introduced in (Bachman 1969) is often used to represent the coset interconnections in a network schema. This notation, called a Bachman

7.0 Transforming the ALF Statement

7.1 Overview of Section 7

The following section will describe the translation algorithms and demonstrate their validity. First the model for query evaluation in ALF initially given in Section 5.2 will be specified. Secondly a model for network databases using an equivalent structure called a D-graph will be described. Thirdly a graphical pattern, called a V-graph, after a similar structure defined in (Palermo 1974), will be derived from each query, and matching this pattern against the database D-graph will be shown to be equivalent to the original model for query evaluation. Fourthly, transformations will be defined on the V-graph, and it will be shown that each transformation produces a V-graph equivalent to the previous one in the sense that the set of answer tuples produced by matching it against the D-graph is unchanged by the transformation. Lastly it will be shown that the pattern matching process corresponds to CODASYL network traversing algorithms.

The translation algorithm may be divided into the following steps:

1) Input and preprocess the ALF statement.

2) Remove virtual attributes from each query.

3) Coalesce equivalent V-graph nodes.

4) Amalgamate equivalent queries.

5) Process each V-graph, starting at the V-graph corresponding to the root query of the Q-graph.

6) Optimise the generated code.

7) Either interpret the optimised code, or use it to generate statements in the required target language.

In step 1, the ALF statement is parsed and put into an internal form. Before the main part of the statement translation, two preprocessing steps are done.

The first of these converts each occurrence of the aggregate function ALL into an occurrence of the EXISTS function. This is done by applying the identity

ALL X WHERE B(X,..) <=> NOT EXISTS X WHERE NOT B(X,..)

to each subquery acted on by an ALL function.

The second preprocessing step removes references to virtual relations, or views. (See Section 9.4)

Step 5, processing the query, is carried out for each subquery as well. This step may be divided as follows:

5a) Find a starting node in the V-graph and an initial access method.

5b) Determine node accessing order and access methods for each node.

5c) Generate intermediate code for the query (This step will involve the processing of subqueries)

If the V-graph is disconnected, steps 5a and 5b will be repeated for each disconnected subgraph.

7.2 Query Evaluation Model

As described in Section 5, a query in ALF is an object of the form

targetlist WHERE *qualification*

Evaluation of a query is a process which produces a set of targetlist tuples. If the query is part of a subquery, then these tuples will be operated on by a aggregate function to produce a scalar value, and if it is part of a command, they will be operated on by the command to produce some sort of output.

As described in Section 5, an ALF query may be thought of as being evaluated by a four stage process.

1) If LV_1, ... ,LV_n are the local variables in a query, and R_1, ... ,R_n are their respective ranges, take the cartesian product of the relations R_1 ... R_n. If there are global variables GV_1, ... GV_m in the same query, then each time the query is executed, each global variable will have as value a tuple from a relation in a higher level query. These tuples TGV_1 ... TGV_m are concatenated to each tuple of the cartesian product.

Thus, the cartesian product has the following form:

```
        LV1                   LV2                   GV1
  +------------+      +--------------+  . . .  +----------+  . . .
  |   R1[1]    |      |    R2[1]     |              TGV1
  |   R1[2]    |  X   |    R2[2]     |    X                    X
  |   R1[3]    |      |    R2[3]     |
  |     :      |      |      :       |                :
```

As the cartesian product contains every possible combination of tuples from its component local variable relations, the cardinality of the cartesian product will be the product of the cardinalities of each local variable relation. Materialising such a huge relation is out of the question, for efficiency reasons.

2) The query qualification is evaluated using attribute values from each tuple in the cartesian product. As the qualification is a boolean expression of terms containing items of the form variable.attribute, each column in the cartesian product must be labelled implicitly with the attribute name and the relation variable from which it was derived. All tuples for which the qualification is false are rejected, and all for which the qualificiation is true are retained.

3) The remaining tuples are projected onto those items which occur in the query targetlist, that is, items not participating in the targetlist are ignored.

4) The targetlist expressions are computed from the items in the tuples that remain.

5) If the query occurs as part of a subquery, the aggregate function of the subquery is applied to the resulting tuples, and if the query occurs as part of a retrieval statement, the statement command is applied to the resulting tuples.

There are several noteworthy features of this algorithm.

The first is that the process contained no operation occuring between tuples of the cartesian product, and so if each tuple of the cartesian product could be materialised one at a time, the qualification could be evaluated and that tuple accepted or rejected before materialising the next tuple.

The second is that if the cartesian product tuples could be produced incrementally in the sense that only one extra relation was multiplied into the cartesian product at once, and if the qualification could be factored into conjuncts of the form Q = *factor* AND *residue*, where *factor* contained items from those parts of the tuple already materialised, then further relations need only be multipled into the cartesian product if the part so far materialised satisfied the boolean expression *factor*.

The most obvious case of this is where tuples from a single relation would not enter the cartesian product if they did not satisfy a conjunct in the qualification referring only to a relation variable with that relation as range.

7.3 D-graph

The underlying network database corresponds structurally to a graph in which record instances are represented as nodes, and the fact that record instance RM is a member of the coset S1 whose owner record instance is the record RO is represented by an arc, labelled with the coset name, pointing from RO to RM. As each coset instance is an instance of an owner record, associated with a set (in the set theoretic, rather than CODASYL sense) of member records, each coset occurrence corresponds structurally to a graph in the form shown in Figure 7.1.

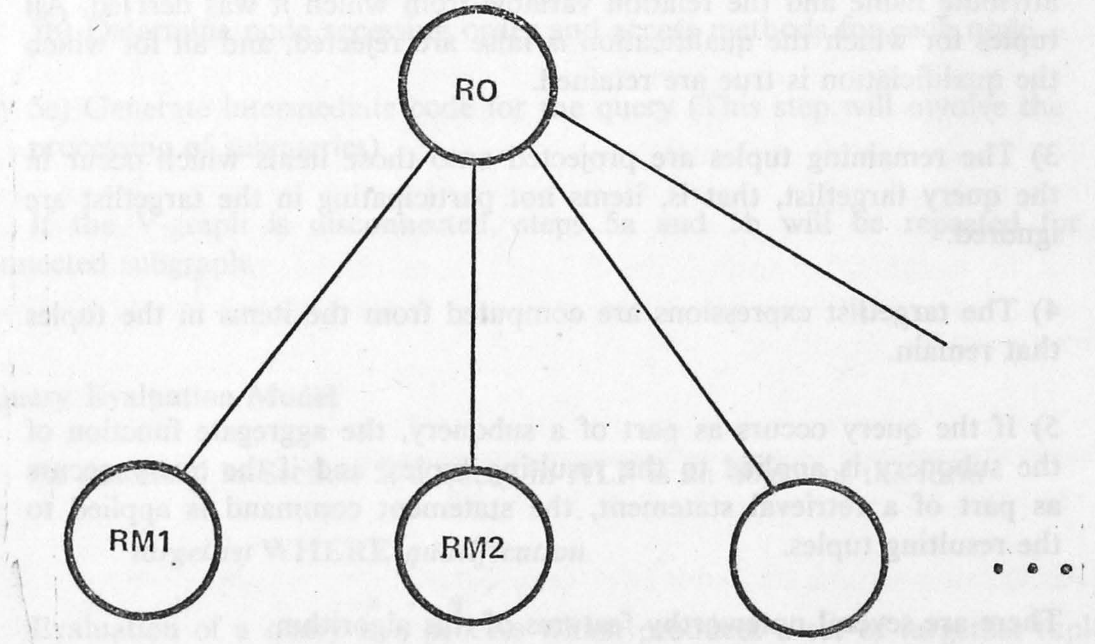


Fig. 7.1 A coset Instance

The coset members may be considered as being ordered from left to right. This ordering may be arbitrary, or may correspond to the ordering defined in the CODASYL schema specification. The whole database viewed in this way will be called the D-graph of the database. This D-graph structure in no way implies any particular physical implementation.

Operations on this graph corresponding to CODASYL DML operations may be readily defined assuming a coset ordering in the D-graph; in particular finding the first record in a coset occurrence is equivalent to finding the leftmost member node in the coset occurrence in the D-graph, finding the next record in a set occurrence is equivalent to moving from the current member to the one on its right in the D-graph, and finding the owner is equivalent to following the arrow representing the coset backwards from a member to the owner in the D-graph.

7.4 Pattern Matching

Pattern Matching is a process which arises in a wide variety of situations throughout computing. Problems in syntax analysis, text processing, higher level language control structures, and picture processing can all be formulated using the pattern matching paradigm. There are numerous discussions of different aspects of the pattern matching process in the literature, for example Vol I of (Aho 1972), Chapter 8 of (Waite 1973), (Sussman 1970), (Gimpel 1973), (Griswold 1968), (Miller 1968), (Bobrow 1974), (Hewitt 1972) and (Rulifson 1972). A general formulation is to regard it as a process with the following properties:

1) There is a structure of some sort called the *pattern*, containing constant parts and nonconstant parts, or *unbound variables.*

2) There is a structure of a similar sort containing constant parts only. This will be called the *subject* of the pattern matching process.

3) Values must be assigned to the unbound variables of the pattern so as to make it equivalent in some way to a part of the subject. In the case where there are no variables in the pattern, the pattern must itself be equivalent to part of the subject (for example searching for a substring in a longer string of text in a text editor). *Equivalence* in pattern matching may be more than simple equality; in addition to some structural equivalence between the pattern and the subject part, there may be some other condition which must also be satisfied for the match to succeed. This other condition may be specified procedurally, or as a formula in a logical calculus.

In a practical implementation of a pattern matching algorithm, the pattern is not matched with the subject in one hit, but is matched incrementally. A fragment of the pattern is selected and a fragment of the subject matching that fragment is found. A pattern *cursor* is moved onto a further fragment of the pattern, a corresponding data cursor moved onto the next appropriate fragment of the subject, and that fragment tested against the fragment of the subject. If these fragments matched, then the process continues until the pattern had been completely used. If these fragments did not match, the match would fail and fall back to the previous stage, where the pattern fragment would be tried against a further fragment of the subject. For example, in matching a short text string against a longer subject string in the most obvious (although not the most efficient) way, the first character of the pattern is tested against successive characters of the subject string until equality is found. Then the next character of the pattern is tested against the next character of the subject. If it is equal, the process continues, if not, then another attempt is made to match the first character.

This process could be compiled into a procedure consisting of a set of nested loops, in which each loop was responsible for all the matchings of a single unbound pattern variable. The outermost loop would generate all the matches for the initially selected pattern variable; the next loop in the nest would generate all matches for its pattern variable given the constraints imposed by the binding of the initially selected variable; the next loop would generate all matches for its pattern variable given the bindings for the variables in the outer loops, and so on. The complete matching would be obtained inside the innermost loop. The loops would have the following structure :

```
WHILE furthur bindings for $V_1$ are available DO
assign a furthur binding for $V_1$
IF binding for $V_1$ is OK THEN
WHILE furthur bindings for $V_2$ are available DO
assign a furthur binding for $V_2$
IF binding for $V_2$ is OK THEN
WHILE ...
:

:
ENDWHILE
ENDIF
ENDWHILE
ENDIF
ENDWHILE
```

7.5 V-graph

Keeping this general pattern matching process in mind, the *initial V-graph* of each query is defined as being a set of nodes labelled with the local variables of the query. The graph at this stage contains no arcs. It will play the role of a pattern in a pattern matching process similar to that described above, the nodes playing the role of the unbound variables of the V-graph pattern. The cartesian product generation involved in the model for query evaluation described in 7.2, involves generating every combination of tuples for the local variable relations. This product could be generated by matching the V-graph in every possible way against the database D-graph, so that each V-graph node matches a D-graph node in the range of the variable represented by the node. (For brevity, the terms node and variable will be used somewhat interchangeably where no confusion arises, and the range of the variable represented by a V-graph node will be called the range of the node).

In each match of the V-graph against the D-graph, each node variable is instantiated with the tuple from the matched record (that is, the D-graph node), and the query qualification evaluated. If the qualification was true, the answer tuple represented by the match would be accepted, and either used in the computation of an aggregate function, or output.

7.6 V-graph transformations

Matching the initial V-graph against the D-graph materialises all tuples in the cartesian product without taking into account any of the constraints which are implied by join terms in the query qualification. The initial V-graph may be transformed to take these constraints into account. The qualification will itself be transformed in the process. The V-graph obtained by the processes described in this section will be more suitable for the generation of network accessing programs, in that it will inhibit the generation of cartesian product tuples which cannot appear in the result.

7.6.1 Remove Virtual Attributes

In general the qualification will contain virtual attributes, which must be materialised before the qualification can be evaluated. They are materialised as described in section 6, by using a coset instance and obtaining the data item corresponding to the source attribute from the owner record instance of the coset.

When the source attribute is itself virtual, this process must be repeated. This process may continue up a hierarchy until an actual source attribute is reached.

This materialisation, which corresponds to step 2 in the translation process, is made explicit in the V-graph by repeating the following procedure until no more virtual attributes occur in the query.

For each query, the following process is performed.

For each virtual attribute in the form R.DV where R is local in the current query, and R.DV is either present in the targetlist or the qualification or is an argument of a lower order query, create a new node in the V-graph of the current query. This node will be labelled with a new relation variable, say NEWV, whose range is the relation containing the source attribute for the virtual attribute R.DV. An arc, labelled with the coset used to materialise the virtual data item, is created and points from node NEWV to node R. All occurrences of the virtual item R.DV in the qualification and targetlist of the query, and in subqueries containing R.DV as an argument should be changed to NEWV.DS, where DS is the source attribute.

As a simple example, assume a schema as shown in Figure 7.2. In this schema, relation EMP contains a virtual attribute D#, which is materialised from the DEPT relation using the coset ESET.

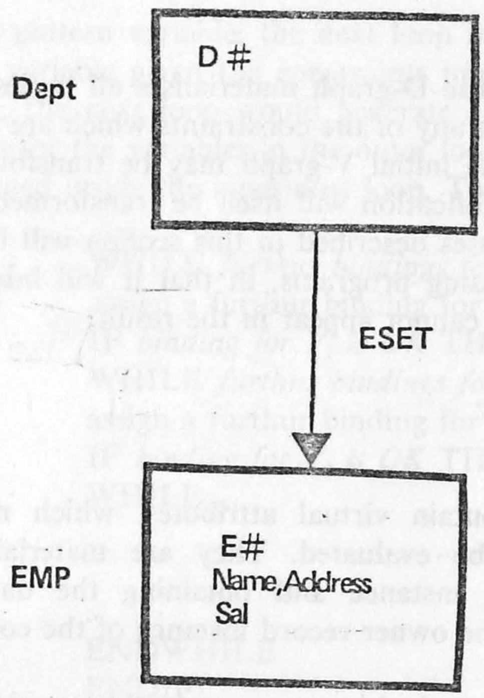


Fig. 7.2 Network Schema

A trivial ALF query using this schema follows:

```
OUTPUT EMP.NAME, EMP.ADDRESS
WHERE EMP.D# EQ "ABC"
```

The initial V-graph would consist of the single node EMP (Figure 7.3)

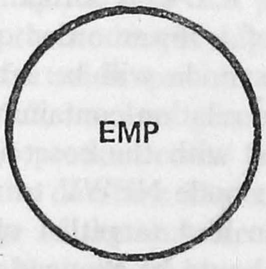


Fig. 7.3 Initial V-graph

The only virtual item in the query is EMP.D# as D# is a virtual attribute of EMP. After removal of this virtual item, the query graph would have the form shown in Figure 7.4.

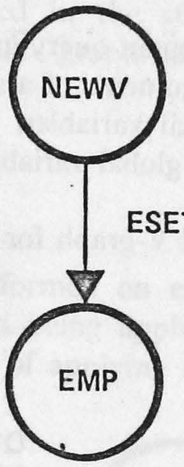


Fig. 7.4 V-graph after virtual item removal

The transformed query would look like this

OUTPUT EMP.NAME,EMP.ADDRESS WHERE NEWV.D# EQ "ABC".

Thus the V-graph represents the coset connections to be used in making all the virtual attributes explicit.

Another more complicated example may help make this process clearer.

Using the same schema, consider the statements

RANGE OF EMP1 IS EMP.
OUTPUT EMP.NAME, EMP.ADDRESS WHERE
EMP.SAL GT AVG (EMP1.SAL WHERE EMP1.D# EQ EMP.D#).

that is, Give names and addresses for all employees who earn more than the average salary for their department.

The Q-graph for this retrieval statement has two nodes (Figure 7.5)

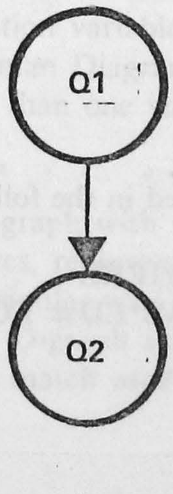


Fig. 7.5 Q-graph for sample query

Q1 is the main query and Q2 is the query acted upon by the aggregate function AVG. As the arguments of a subquery are those items used in the subquery which are qualified by global variables, the (single) argument of Q2 is EMP.D#. In this case EMP is the only global variable in Q2.

The initial V-graph for Q1 is as shown in Figure 7.6.

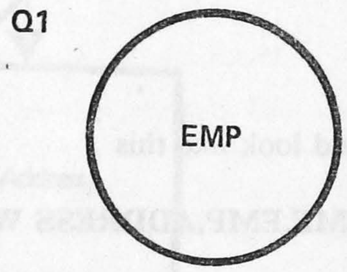


Fig. 7.6 Initial V-graph for Q1

This is transformed into the structure shown in Figure 7.7 when the virtual data item EMP.D# is removed.

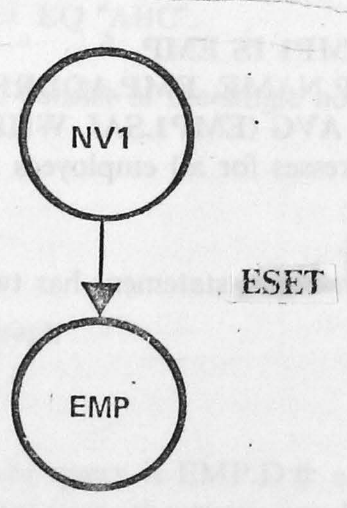


Fig. 7.7 After removal of EMP. D #

At this stage, the ALF command has been transformed in the following way.

OUTPUT EMP.NAME, EMP.ADDRESS WHERE
EMP.SAL GT AVG(EMP1.SAL WHERE EMP1.D# EQ NV1.D#).

The actual item NV1.D# has been substituted in the statement for each occurrence of the virtual item EMP.D#, and the initial V-graph has been augmented to show this constraint. Implicitly, the range command

RANGE OF NV1 IS DEPT.

has been issued.

This virtual attribute removal algorithm is performed on each query in the Q-graph, starting at the root node of the Q-graph and being applied in a recursive fashion to the other queries in the Q-graph. The result of applying it to Q2 is shown in Figure 7.8.

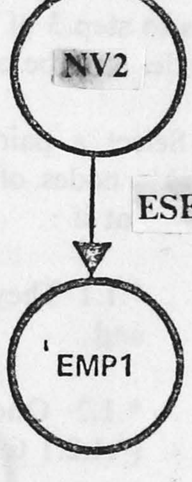


Fig. 7.8 Remove EMP1. D# in Q2

The query at this stage is

OUTPUT EMP.NAME, EMP.ADDRESS WHERE EMP.SAL GT AVG(EMP1.SAL WHERE NV2.D# EQ NV1.D#).

The source attribute introduced during this removal process may itself be a virtual attribute. If it is, the process is repeated until the final source attribure is non-virtual, that is, actually stored in the relation. In the example of 4.1.3, removal of references to EMP.C# would involve two levels in the hierarchy.

Each query has now been split into two parts; one part consisting of the original query with actual attributes substituted for the original virtual attributes, and the other part consisting of a graphical representation of the access paths used for obtaining these actual data items. In queries where each relation is represented by at most one relation variable, the V-graph will be topologically a subset of the C-graph (that is, Bachman Diagram) for the schema, but in queries where a relation is ranged over by more than one variable, such as the parts explosion example, this will not be the case.

The V-graph with virtual items removed may be thought of as a pattern, with the labelled arcs, representing cosets, being constant, and may be matched against the D-graph in a similar way to the initial V-graph. V-graph nodes in the pattern must be matched with D-graph nodes of records in their range, and in addition arcs in the V-graph must match arcs in the D-graph. From the definition of the way in which virtual attributes are materialsied, it is evident that this matching will produce the same set of tuples as produced by the original matching followed by materialisation of virtual attributes.

7.6.2 Coalesce equivalent nodes

A further set of transformations may be applied to the V-graph. For each matching of the V-graph with the D-graph, it may be shown that certain pairs of nodes in the V-graph will always match the same node in the D-graph. These pairs of equivalent nodes may be coalesced, thus simplifying the V-graph. This procedure corresponds to step 3 of 7.1. The following two steps (*.1 and *.2) are performed until no more nodes may be coalesced:

*.1 Select a pair of equivalent nodes to be coalesced from the local variable nodes of the query. Let them be RI and RJ. RI and RJ are equivalent if :

*.1.1 They both range over the same relation, say R, and

.1.2 One of the following three conditions holds (.1.2.1 to *.1.2.3):

*.1.2.1 For some candidate key of R, say $D_1,...,D_n$, and a logical expression of the form

$$\begin{aligned} & RI.D_1 = RJ.D_1 \\ \text{AND } & RI.D_2 = RJ.D_2 \\ & \vdots \\ \text{AND } & RI.D_n = RJ.D_n \end{aligned}$$

can be factored out of the query qualification. That is, the qualification specifies that the candidate key be equal in the tuples matched by RI and RJ.

*.1.2.2 The structure shown in Figure 7.9(a) occurs in the the V-graph.

*.1.2.3 The structure shown in Figure 7.9(b) occurs in the V-graph, and the qualification specifies that for some candidate key of the relation ranged over by RI, the attributes of the candidate key not virtually materialised using coset S1 are equal in RI and RJ. Equality means that the same factorisation done in *.1.2.1 can be done here.

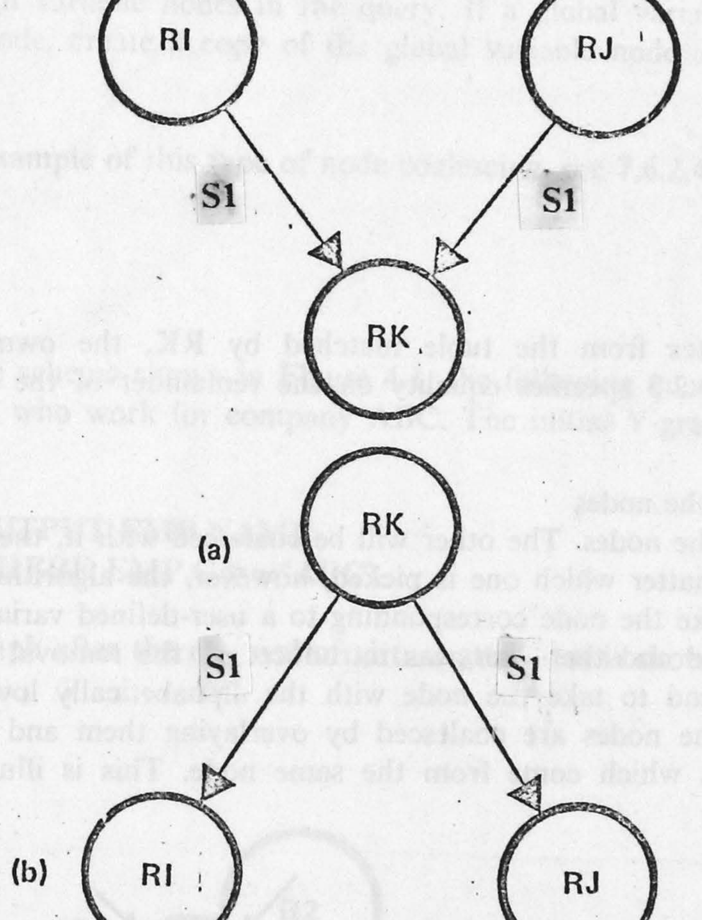


Fig. 7.9

In the case *.1.2.1, for each matching of the V-graph, RI and RJ will always match the same record in the D-graph, as the candidate key uniquely identifies the tuple (record) and it is specified as being equal for RI and RJ.

In case *.1.2.2, matching this substructure of the query graph against the D-graph would involve matching RK with some D-graph node. The node matching RK would be an instance of a member record of coset S1. Nodes RI and RJ would then necessarily both match the owner instance of that instance of S1, as a coset instance has only one owner record.

Case *.1.2.3 is really a special case of *.1.2.1, in which equality of candidate keys occurs. Equality of the components of the candidate keys virtually materialised through S1 occurs because RI and RJ both obtain those attributes from the tuple matched by RK, the owner of S1. Condition *.1.2.3 specifies equality on the remainder of the candidate key attributes.

*.2 Coalesce the nodes
Pick one of the nodes. The other will be coalesced with it, then deleted. It does not matter which one is picked, however, the algorithm used in ALF is to take the node corresponding to a user-defined variable if one is user-defined and the other was introduced in the removal of virtual data items, and to take the node with the alphabetically lowest name otherwise. The nodes are coalesced by overlaying them and removing repeated arcs which come from the same node. This is illustrated in Figure 7.10.

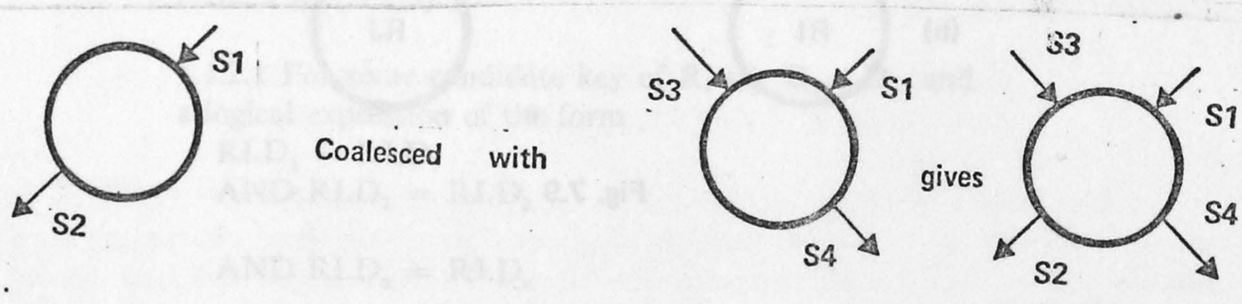


Fig. 7.10

If node RJ is coalesced with RI, delete the node RJ and all arcs entering or leaving it.

As the last step in coalescing two nodes (*.2), substitute RI for all occurrences of RJ in the query being processed, and in all subqueries of that query which have RJ as a global variable.

RI will then be a global variable of each such subquery. So far, nodes which have been coalesced have both been members of the same V-graph.

The value of a subquery is determined by the values of the global variables when the subquery is evaluated. Values of these global variables are fixed during an evaluation of the subquery. A local variable in a subquery may be equivalent to a global variable by *.1.2.1, (equality of candidate keys). In this case, that local variable would be fixed during the evaluation of the subquery. This leads to the following additional process.

For each query, repeat the process of coalescing equivalent nodes described in *.1 and *.2, testing the equivalence of each global variable node in the query with each of the local variable nodes in the query. If a global variable is equivalent to a local variable node, create a copy of the global variable node and coalesce the local node with it.

For an example of this type of node coalescing, see 7.6.2.4.

Example 7.6.2.1

Using the schema shown in Figure 4.1, the following query asks for the names of all employees who work for company ABC. The initial V-graph is the single node EMP.

OUTPUT EMP.NAME
WHERE EMP.C#="ABC".

The V-graph after the removal of virtual attributes is shown in Fig 7.11.

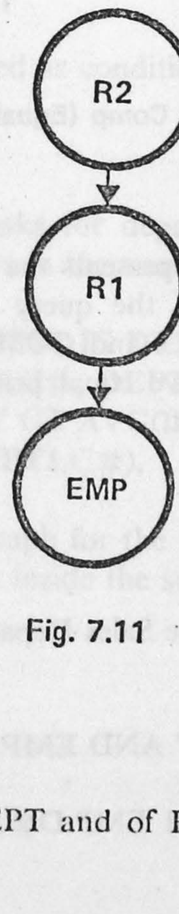


Fig. 7.11

The range of R1 is DEPT and of R2, COMP. No nodes can be coalesced in this query.

Example 7.6.2.2

The following query, which also uses the schema of Figure 4.1, asks for departments of companies whose headquarters are in the ACT.

OUTPUT DEPT.NAME WHERE
DEPT.C# = COMP.C# AND COMP.CLOC = "ACT".

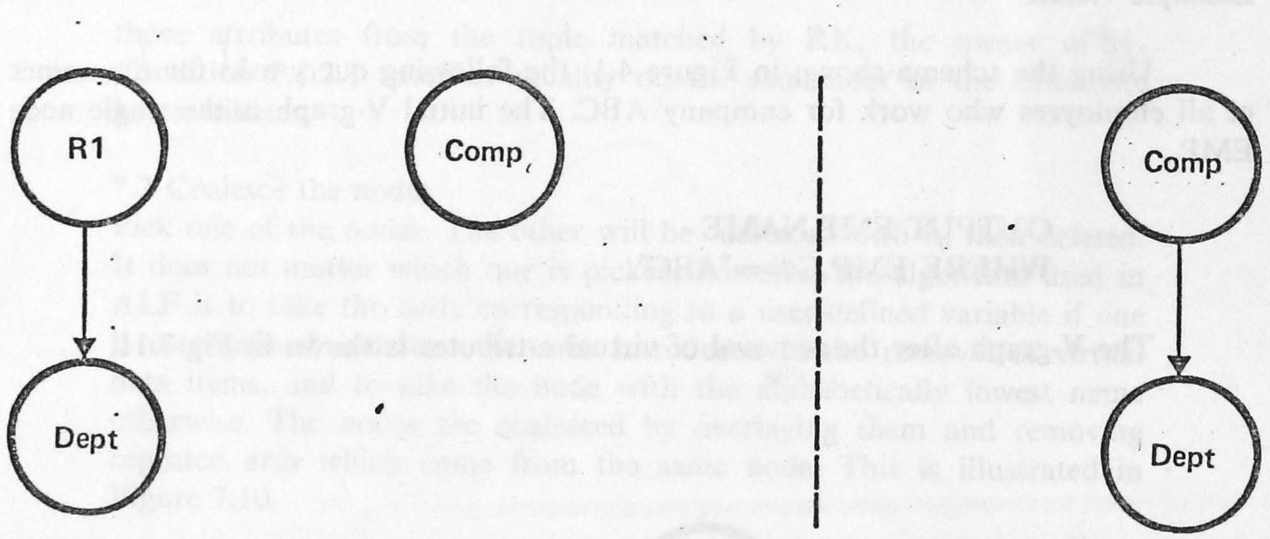


Fig. 7.12 Coalesce R1 and Comp (Equal Primary Keys)

The left hand graph in Figure 7.12 represents the situation after removal of the virtual attribute DEPT.C#. At this stage the query contains a conjunctive term R1.C# = COMP.C#. As the variables R1 and COMP have the same range, (the COMP relation), and as these variables have equal primary keys (condition *.1.2.1) they may be coalesced as shown on the right.

Example 7.6.2.3

Output the names of employees of the Sales Department of Company ABC.

OUTPUT EMP.NAME
WHERE EMP.C# = "ABC" AND EMP.D# = "SALES".

The virtual items are EMP.C# and EMP.D#. After removal of these, the situation on the left of Figure 7.13 exists.

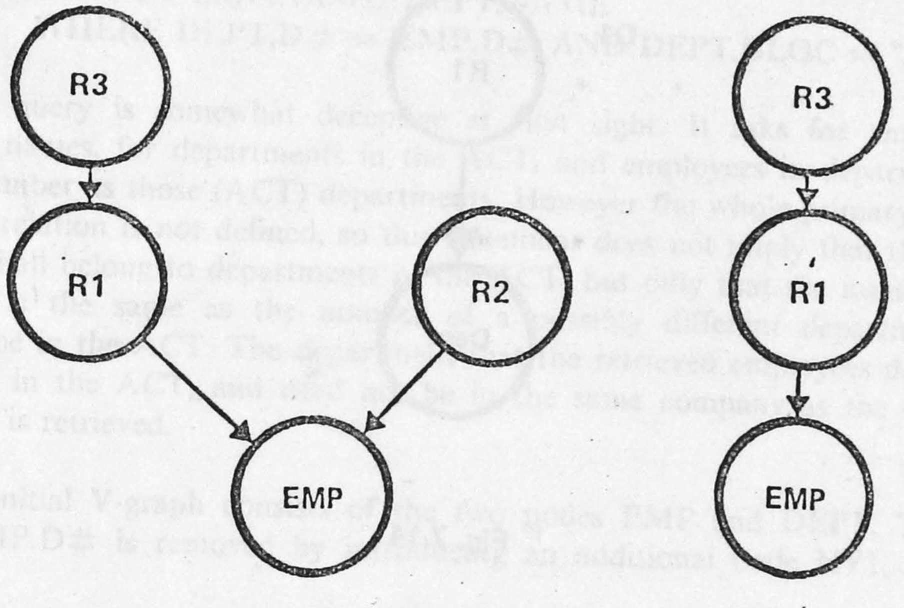


Fig. 7.13 Coalesce R1 and R2

R1 and R2 are coalesced as condition *.1.2.2 is satisfied.

Example 7.6.2.4

This ALF statement asks for departments whose budget is greater than the average budget for departments in the same company.

```
RANGE OF DEPT1 IS DEPT.
OUTPUT DEPT.C#, DEPT.D# WHERE
DEPT.BUDGET GT AVG(DEPT1.BUDGET WHERE
DEPT.C#=DEPT1.C#).
```

Fig 7.14 shows the V-graph for the root query after removal of the virtual data item DEPT.C# (which occurs inside the subquery).

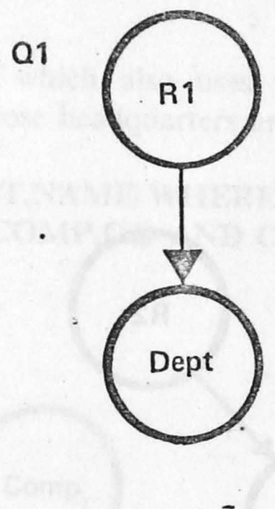


Fig. 7.14

Figure 7.15 (left side) shows the V-graph for Q2, the query acted on by the aggregate function AVG.

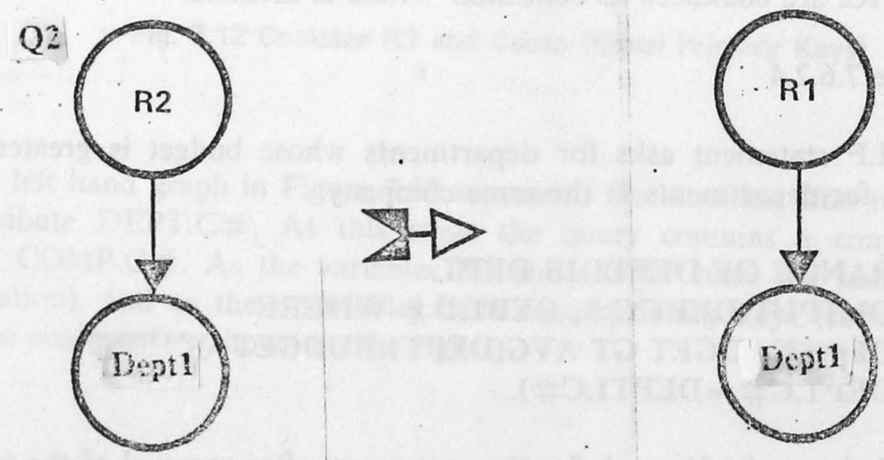


Fig. 7.15

In this case, R2 in the V-graph for Q2 is coalesced with a copy of R1 which is inherited from the higher level query Q1. This is shown on the right of 7.15.

Example 7.6.2.5

OUTPUT EMP.NAME, DEPT.NAME
WHERE DEPT.D# = EMP.D# AND DEPT.DLOC = "ACT".

This query is somewhat deceptive at first sight. It asks for employee and department names, for departments in the ACT, and employees in departments with the same number as those (ACT) departments. However the whole primary key of the department relation is not defined, so this statement does not imply that the retrieved employees shall belong to departments in the ACT, but only that the number of their department is the same as the number of a possibly different department which happens to be in the ACT. The department that the retrieved employees do belong to need not be in the ACT, and need not be in the same company as the department whose name is retrieved.

The initial V-graph consists of the two nodes EMP and DEPT. The virtual attribute EMP.D# is removed by introducing an additional node NV1, as in FIG 7.16.

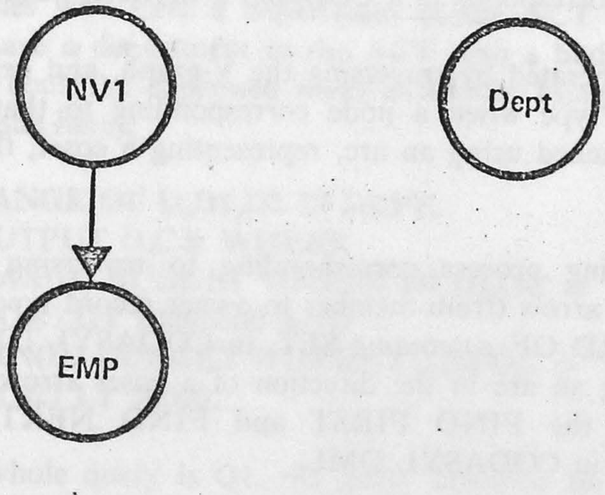


Fig. 7.16

After removal of the virtual attributes, the command has the form

OUTPUT EMP.NAME,DEPT.NAME WHERE DEPT.D# = NV1.D# AND DEPT.DLOC = "ACT".

This does not allow nodes DEPT and NV1 to be coalesced, as equality has not been specified on the whole of the primary key of the DEPT relation. The query, as it stands, will be processed correctly.

7.7 Use of V-graph

Sections 7.4 and 7.5 described a general pattern matching model, and introduced the idea that the V-graph should be used as a pattern and matched against the database, thought of as a D-graph. A procedure, analogous to the one in 7.4, containing code to navigate over a network database using DML commands, is generated by the ALF translator.

A node in the V-graph is selected as a start node. This node is matched with a D-graph node in its range. Then an arc to or from that V-graph node is matched with a corresponding D-graph arc, and the cursor of the matching process passes to the node on the end of that pattern arc. The subpattern following that arc must be matched against the D-graph before the matching control returns to the originally matched node to match the parts of the pattern starting with the other arcs coming from that node. After all parts of a pattern or subpattern have been matched, the process fails back to the last decision point to take the next alternative, thus finding all matches. For any V-graph, a procedure implementing this process may be generated. This procedure corresponds to a CODASYL network traversal procedure.

The procedure is generated by traversing the V-graph, and generating code to access a particular record type when a node corresponding to that record type is visited. If the node was reached using an arc, representing a coset, then that coset is used to access the record.

The pattern matching process corresponding to traversing a V-graph arc against the direction of the arrow (from member to owner record type) corresponds to a FIND OWNER RECORD OF *cosetname* SET, in CODASYL DML. The process corresponding to traversing an arc in the direction of a coset arrow (from owner to member) corresponds to the FIND FIRST and FIND NEXT RECORD IN *cosetname* SET commands in CODASYL DML.

7.8 Amalgamation of Equivalent Queries

In ALF commands containing two or more aggregate functions, it may be possible to compute more than one of the aggregates with a single pass through the relevant records. This is in fact the case in the example in 4.1.8. ALF detects such cases where two or more queries are structurally similar. Two queries may be amalgamated to be evaluated together if they satisfy the following two conditions.

1) The V-graphs must be able to be unified.
Unification can be thought of as two-sided pattern matching, in which there are two pattern data structures, each containing unbound variables. Values must be assigned to the unbound variables so that the two patterns are equal. In the case of two V-graphs the role of unbound variables is played by local variable nodes in each graph. The two graphs are unifiable if the nodes in each graph may be paired in such a

way that the nodes in each pair are equivalent with respect to unification. Two nodes are equivalent with respect to unification if either they both represent the same global variable, in which case they must be paired in the unification, or they have the same range, identical sets of inpointing and outpointing arcs, and equivalent nodes at the ends of the arcs.

Informally, the two graphs are unifiable if one may be laid completely across the top of the other so that the arcs and global variable nodes of one are matched by global variable nodes of the other, the remainder of nodes put into correspondence have the same ranges, and all arcs in each graph match in name and direction.

2) The qualifications of the queries associated with each V-graph must be equivalent (that is, identical up to associativity), after the nodes in the first have been substituted for the equivalent nodes in the second.

As one example of this process, let us modify the statement of 4.1.8 as follows - "Find companies who have a department in the ACT with a budget of less than 30000 or who have a department in the ACT with a budget greater than 10000000." (This statement could be expressed more efficiently using the EXISTS function, but this is not the issue here).

```
RANGE OF D,D1,D2 IS DEPT.
OUTPUT D.C# WHERE
MAX(D1.BUDGET WHERE D1.DLOC = "ACT" AND D1.C# =
D.C#) GT 10000000 AND
MIN(D2.BUDGET WHERE D2.DLOC = "ACT" AND D2.C# =
D.C#) LT 30000.
```

Say the whole query is Q1, the query operated on by the aggregate MAX is Q2, and the query operated on by the aggregate MIN is Q3. The final state of each of the graphs after virtual attribute removal and coalescing equivalent nodes is shown in Figure 7.17.

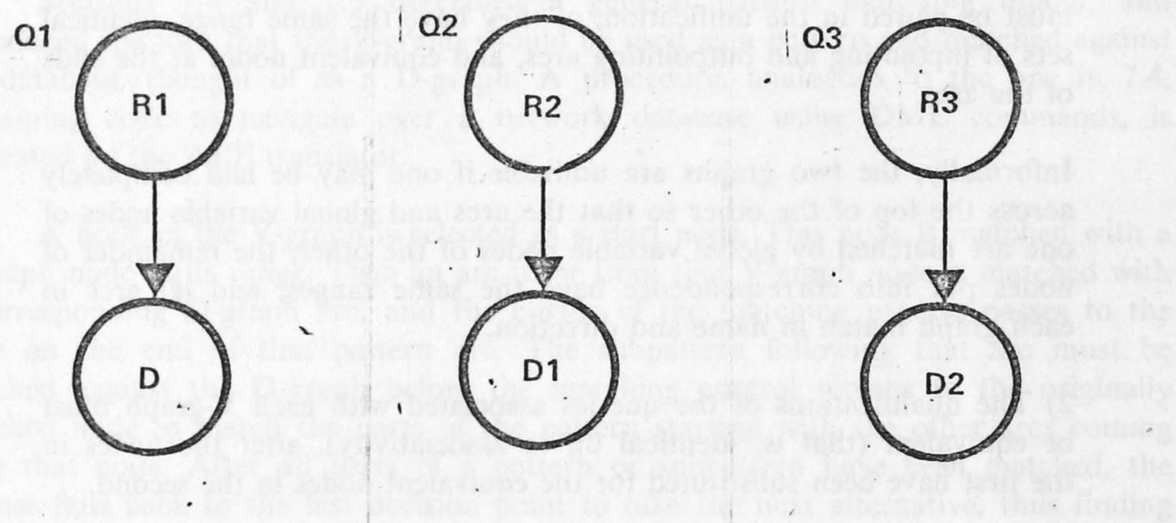


Fig. 7.17

R1, R2 and R3 have COMP as range

Queries Q1, Q2 and Q3 have the following form at this stage:

Q1: R1.C# WHERE MAX(Q2(R1.C#) GT 10000000 AND MIN(Q3(R1.C#)) LT 30000.

Q2: D1.BUDGET WHERE D1.DLOC="ACT" AND R2.C#=R1.C#

Q3: D2.BUDGET WHERE D2.DLOC="ACT" AND R3.C#=R1.C#

Each of the three V-graphs can be unified with any other, however the qualification of Q1 is not equivalent to either of the other two qualifications after substitution of equivalent nodes. In Q2 and Q3, equivalent node pairs are (R2,R3) and (D1,D2). Equivalent expressions are obtained when R2 and D1 from query Q2 are substituted for R3 and D2 in query Q3. Thus the graphs for Q2 and Q3 may be unified, and the aggregates which operate on Q2 and Q3, MAX and MIN, may be evaluated together.

What this condition really says is that a sufficient condition for two aggregate functions to be evaluated together is that the same path through the network is traversed by both, and that exactly the same set of tuples is retrieved by both. that the same path is traversed is implied by the identical structure of the V-graphs, and that the same set of tuples would be retrieved while traversing each path is implied by the equivalence of the two qualifications. In fact both these conditions are far too harsh. Provided that a reasonable amount of the path through the CODASYL network is common to both queries, it should be possible to process that part of them together. Also even if the tuples retrieved by both queries, as determined by their respective qualifications, are not the same it still should be possible to process them together. This sort of optimisation has not been pursued in the current ALF system.

An additional sort of optimisation which has not been pursued in the current ALF system is to take into consideration the wider context in which an aggregate function occurs, so that a complete iteration through all the tuples in the targetlist of the subquery might be avoided.

As an example, take the case where SUM(...) GT 10000 occurs conjunctively in a qualification. The computation of the query operated on by SUM could be terminated as soon as the sum became greater than 10000 by moving the test inside the subquery computation. COUNT, MAX and MIN are other candidates for this treatment, although AVG and SD must necessaily process all the tuples.

A subquery containing the aggregate function EXISTS must not be coalesced with any other subquery. This is because the EXISTS function ceases iteration as soon as one tuple is found. (See Section 7.1) the same set of tuples would be retrieved while traversing each path is implied by the equivalence of the two qualifications. In fact both these conditions are far too harsh. Provided that a reasonable amount of the path through the CODASYL network is common to both queries, it should be possible to process that part of them together. Also even if the tuples retrieved by both queries, as determined by their respective qualifications, are not the same it still should be possible to process them together. This sort of optimisation has not been pursued in the current ALP system.

An additional sort of optimisation which has not been pursued in the current ALP system is to take into consideration the wider context in which an aggregate function occurs, so that a complete iteration through all the tuples of the targetlist of the subquery might be avoided.

As an example take the case where SUM(...) GT 10000 occurs conjunctively in a qualification. The computation of the query operated on by SUM could be terminated as soon as the sum became greater than 10000 by moving the test inside the subquery computation. COUNT, MAX and MIN are other candidates for this treatment, although AVG and SD must necessarily process all the tuples.

A subquery containing the aggregate function EXISTS must not be coalesced with any other subquery. This is because the EXISTS function ceases iteration as soon as one tuple is found. (See Section 7.1)

R1, R2 and R3 have COMP as range.

Queries Q1, Q2 and Q3 have the following form at this stage:

Q1: R1.C# WHERE MAX(Q2(R1.C#)) GT 1000000 AND MIN(Q3(R1.C#)) LT 30000.

Q2: D1.BUDGET WHERE D1.DLOC='ACT' AND R2.C#=R1.C#

Q3: D2.BUDGET WHERE D2.DLOC='ACT' AND R3.C#=R1.C#

Each of the three V-graphs can be unified with any other, however the qualification of Q1 is not equivalent to either of the other two qualifications after substitution of equivalent nodes. In Q2 and Q3, equivalent node pairs are (R2,R3) and (D1,D2). Equivalent expressions are obtained when R2 and D1 from query Q2 are substituted for R3 and D2 in query Q3. Thus the graphs for Q2 and Q3 may be unified, and the aggregates which operate on Q2 and Q3, MAX and MIN, may be evaluated together.

What this condition really says is that a sufficient condition for two aggregate functions to be evaluated together is that the same path through the network is traversed by both, and consequently the same set of tuples is retrieved by both; that the same path is traversed is specified by the identical structure of the V-graphs, and that

8.0 Code Generation

This section describes the generation of intermediate code from the transformed V-graph, targetlist and qualification, and the use to which the intermediate code is put. The intermediate code is described in Appendix B.

In the following sections, some knowledge of the CODASYL specifications as described in (Codasyl 1971) will be assumed. In particular the various forms of the Data Manipulation Language (DML) FIND statement will be referred to.

8.1 Efficiency Considerations

In the Introduction it was stated that the goal of the ALF translator was to produce programs that were efficient. I will make this more explicit now by stating the following aims which contribute to that efficiency.

8.1.1 Whenever possible, the access paths provided by the existence of cosets should be exploited. The generation of the V-graph, and the transformations on it have been directed to this end.

8.1.2 In traversing these cosets, use should be made if possible of any indexes or search keys defined on the coset.

8.1.3 When appropriate, use should be made of CALC keys and SYSTEM owned cosets.

8.1.4 Unnecessary CODASYL DML operations should be avoided. (see 8.5)

8.1.5 The targetlist tuples should be materialised incrementally, without the use of intermediate files.

8.1.6 Tests to reject potential targetlist tuples should be performed as early as possible, before all the targetlist values have been retrieved. Many tuples could then be rejected with a single test.

8.1.7 Unnecessary iteration should not be performed.

8.1.7.1 For example, if the values of a candidate key of a relation are specified (explicitly or implicity) in a query, and a tuple is found containing these values, then the relation need not be scanned further for an additional tuple with the same values.

8.1.7.2 As a further example, if a sorted coset is being traversed (sorted in ascending order, without loss of generality), and the qualification implicity or explicitly specifies upper limits for the sortkey attributes, then the coset should not be traversed any further than necessary, that is beyond a record with sortkey values greater than the specified upper limits.

8.1.8 Terms in the qualification which involve subqueries are much more expensive to evaluate than others, as each subquery itself involves iteration through relations and accesses to secondary storage. In testing the qualification, the evaluation of subqueries should if possible, be avoided. This goal gives rise to two sorts of optimisation.

8.1.8.1 If a test occurs inside an inner loop, and the test contains one or more subqueries, evaluation of the subqueries should be moved out of the loop if the subquery is loop invariant. Examples 4.1.6 and 4.1.7 show cases where this is done. Thus a subquery should be evaluated as soon as all its arguments are defined properly.

8.1.8.2 In the case where the subquery is not loop invariant, optimisation may still be possible. For example, consider the following ALF statement, assuming that the schema of Figure 4.1 applies.

```
OUTPUT COMP.C# WHERE
COMP.CLOC EQ "ACT" . . . . . . . . . . . . . . . .  (1)
AND
AVG(DEPT.BUDGET WHERE
DEPT.C#=COMP.C#) GT 500000. . . . . . . . . . . .  (2)
```

The statement asks for companies in the ACT with an average departmental budget greater than 500000. If a tuple of the COMP relation failed test (1), that is the company was not in the ACT, then test (2) would not need to be performed.
Similarly, if the logical operator had been OR instead of AND, the success of test (1) would also make test (2) unncessary. The treatment of cases such as this, together with more complex ones, is described in 8.4.

8.2 First pass

The intermediate code is produced in two passes over the V-graph. The first pass, described here, finds an initial entry node and access method to the graph and devises a path through the graph, working out the access methods for each coset in the path. The second pass, described in 8.3, actually produces the code.

The V-graph may not necessarily be connected. It would consist of a number of unconnected subgraphs if the original query contained join terms which did not form part of a coset definition, either because they were not equi-joins, or because primary keys were not fully specified in the joins. When there is more than one subgraph, the most suitable start node for the whole V-graph is found using 8.2.1 and the path through that subgraph found using 8.2.2. Then a start node is chosen from one of the other disconnected subgraphs (using 8.2.1) and 8.2.2 applied again. This process is repeated until a path has been derived which visits every node in the V-graph.

Before describing the first pass, several preliminary concepts must be introduced.

If a number of terms occurring in a boolean expression are singled out, the expression may in general be conjunctively factorised into a *factor* and a *residue*, that is,

expression = *factor* AND *residue*

in such a way that the boolean expression *factor* contains only the terms which were previously singled out. The boolean expression *residue* may also contain those terms, as well as other terms.

The algorithm which performs this factorisation is described in (Hall 1974) and is used in many places in the ALF translator. If the factorisation is not possible, *factor* will equal the boolean constant TRUE, and *residue* will equal *expression*.

I will now define the *local condition* of a node. This is a function of those V-graph nodes which are already processed (that is matched with the D-graph in the query evaluation model). It is also a function of the qualification. Terms defined using processed variables (nodes) only may be identified in the qualification, and the previously mentioned factorisation process carried out, with the following results.

qualification = *local condition* AND *residue*

The boolean expression factored out of the qualification in this process is called the *local condition* of the node, and must be true for the node currently being matched to be accepted as part of the current matching.

A term of the form RI.DI = EXP, where EXP is a constant or a function of processed nodes only, is said to define a value of RI.DI. If such a term occurs as a conjunctive factor in a factorisation of the query qualification, then it may be possible to use this value to improve the search efficiency.

8.2.1 Finding Start Node and Initial Access Method

Recall that the model for this query evaluation process is the matching of a V-graph against the database D-graph. The first V-graph node to be matched is the initial entry into the database for a particular query. This node, SN, called the *start node* of the V-graph, is chosen using the following heuristics.

1) SN should be a global node in the V-graph of the query. If there are no global nodes in the V-graph, one of the local nodes is chosen using the following conditions.

2) There should be no inpointing arcs to SN in the V-graph. This means that SN is either an isolated node or is at the top of a hierarchy.

3) There should be a local condition specified for that node. For the initial entry into the V-graph, no other nodes will have been processed, and hence the factorisation producing the local condition will use terms containing the prospective start node only. This heuristic merely says that there should be some possibility of not having to scan every record corresponding to the start node as variable.

4) One of the following conditions should occur. These all specify search methods built into CODASYL-like systems which allow the number of records retrieved to be reduced. These conditions are designed to exploit *primary access methods*, which are usually used on the initial entry to the database. Condition 4a) allows the exploitation of CALC keys, and conditions 4b) to 4e) allow various properties of SYSTEM owned cosets to be exploited.

a) The local condition defines values or ranges of values for all the CALC keys, if the location mode of the record is CALC.

b) If the record is in a SYSTEM owned coset, the local condition defines values for SORTED INDEXED keys or for SORT KEYS, if the coset is SORTED INDEXED or SORTED.

c) The local condition defines values for one of the sets of SEARCH KEYS which are defined on a SYSTEM owned coset.

d) The local condition defines equalities on any items in a record participating in a SYSTEM owned coset. This would enable a format 6 CODASYL FIND command to be used to scan the coset even though no indexes would be available to assist the scan.

e) The record is a member of a SYSTEM owned coset. In this case, even though the coset would have to be scanned record by record with a format 3 CODASYL FIND command, use of the coset would still probably make this faster than scanning the whole area in which the record resides, especially if the area contained more than one type of record.

If there are V-graph nodes satisfying 1), or 2) and 3), but not satisfying 4a) - 4c), then select one of those nodes as the start node.

If there are no V-graph nodes satisfying conditions 1), 2) or 3), then all nodes in the graph are tested against the conditions in 4).

If all of these heuristics fail, pick any node as the start-node.

In selecting the start node, the initial access method will also be found. If condition 4a) is satisfied, the terms defining the CALC keys are factored out of the local condition, and the residue of that factorisation becomes the local condition. The keys and their values are stored, and the initial access method used will be a format 5 CODASYL FIND command.

If conditions 4b) - 4d) are satisfied, the terms defining the key equalities are factored out of the local condition in the same way as for CALC keys, and the initial access method becomes a format 6 CODASYL FIND command on the SYSTEM owned coset.

If condition 4e) is satisfied then no terms defining key equalities may be factored out of the local condition and a format 3 CODASYL FIND command on the SYSTEM owned coset is used.

If no assistance is provided in finding the record using either CALC keys or SYSTEM owned cosets, then the generated program must iterate through the area using a format 3 CODASYL FIND command on the area.

8.2.2 Subgraph Traversal and Access Method Extraction

This section describes a procedure which traverses the V-graph, extracting information needed by the code generation pass. All the nodes of the V-graph are visited in the order which will eventually correspond to the nesting order of the iterative loops of the generated program. This node order is returned by the procedure, and input to the second pass.

The start node of each subgraph in a V-graph is the first pattern node to be applied to the D-graph. An order must be selected for the other nodes in the subgraph. Initially, call the start node the current node. The cursor of the pattern matching process will point to the current node.

The arcs into the current node, representing cosets for which the range of the node is a coset member, and the outpointing arcs, representing cosets for which the range is the coset owner, are sorted into a processing order. The ordering of the arcs determines where the pattern matching cursor will be moved next, or in CODASYL terms, which coset will be traversed next and which record will be accessed next. In the current version of ALF, this ordering merely puts inpointing arcs before outpointing arcs. This means that the cosets for which the node is a member will be traversed (using FIND OWNER commands), before the cosets for which the node is a member.

The local condition of the current node is computed from the query qualification and the list of already processed (matched) nodes.

The residue after factorising out the local condition becomes the query qualification. In this way, the qualification is eroded away as the graph traversal (pattern matching) process proceeds. The residue erodes to nothing (that is, TRUE) after the last node is processed, as at that stage, with all variables defined, all remaining terms may participate in the factorisation.

For each arc, an access method for the coset it represents is determined, using similar criteria to those used for SYSTEM owned cosets when finding the start node.

If the arc is an inpointing arc, only one access method, a format 4 FIND command "FIND OWNER ..." is available. For outpointing arcs, the following possibilities exist.

1) The local condition defines values for SORTED INDEXED or SORT keys, if the coset is SORTED INDEXED or SORTED.

2) The local condition defines values for one of the sets of SEARCH KEYS defined on the coset.

3) The local condition defines equalities on any other (unindexed) data items in the coset member record.

If any of these possibilities occurs, a format 6 FIND command may be used. The terms defining the relevant data items are factored out of the local condition, and the residue of that factorisation becomes the local condition for the node. In this case the values of the data items are saved, to be set in the *user working area*, (UWA, see (Codasyl 1971)) before execution of the FIND command. Note that the values of data items to be used in Format 6 FIND commands may be constants, or may be expressions using data items from previously found records.

In addition, if the coset is sorted, and upper limits on the sortkeys can be conjunctively factored out of the local condition, these upper limits are saved and used in pass 2 to exit from the coset iteration loop as soon as the corresponding item values from the record exceed them. The residue of this factorisation becomes the local condition for the node. For example if "RI.DI LT value" is a conjunctive term in the local condition, and DI is the leading sortkey of the coset, the coset iteration may be exited as soon as a record with DI greater than or equal to *value* is encountered.

If none of the above possibilities occurs, iteration through the coset must be done using a format 3 FIND statement, one record at a time.

The access methods for the arcs are determined in the previously calculated arc order. After determining the access method for a particular arc, the procedure just described is applied in a recursive manner to the node at the end of that arc. This is done for all arcs except for the one used to access the node initially.

In this way, a path through the whole graph, together with the access methods used for each arc on the path, is determined.

8.3 Second Pass

The second pass of the code generation algorithm takes the path through the V-graph, produced in the first pass, and generates intermediate code from it. Code for the highest level query in the Q-graph is generated first.

8.3.1 Graph Traversal: Code Generation and Subquery Compilation

For each query, the code generated has the following form :

1) Code to initialise query processing.

2) Nested loops to materialise a targetlist tuple.

3) Code to finalise query processing.

If the query was a root query, operated on by one of the top level ALF commands, code inside the innermost loop, (which is executed after a complete targetlist tuple has been materialised), is generated to carry out the command. The initialisation and finalisation code carries out any functions ancilliary to this, for example opening and closing files.

If the query was one operated on by an aggregate function, the initialisation code assigns initial values to the variables used in computing the aggregate function.

Code in the innermost loop, executed when all targetlist items for a single tuple of the query targetlist had been materialised, accumulate the quantities used in computing the aggregate function value, and the finalisation code actually computes the function. For example, in the case where the aggregate function was AVG, the initialisation code sets a counter and a sum variable to zero; the innermost loop code increments the counter and accumulate the value of the targetlist item being averaged in the sum variable; and the finalisation code divides the sum by the counter.

If this query had been amalgamated with others using the procedures described in 7.8, initialisation, accumulation and finalisation for more than one aggregate function would be generated for a single set of nested loops.

There is special treatment for the aggregate function EXISTS. A logical variable representing the value of the EXISTS function is set to FALSE in the query initialisation code. The code generated for the innermost loop, executed when the first tuple satisfying the query has been materialised, consists of setting this logical variable to TRUE, followed by an EXITWHILE out of the outermost WHILE loop of the query. Thus if any tuple satisfies the query, the value of the EXISTS function will be TRUE and no furthur iteration will be performed. If no tuple satisfies the query, the innermost loop code will not be executed and the EXISTS value will remain FALSE. For examples of this, see Appendix E.

The path through the V-graph returned by the first pass is in the form of a list of nodes, and the procedure generated by the second pass is similar to the pattern matching procedure described in 7.6. As previously stated, this procedure contains a set of nested loops, one loop for each node in the V-graph. When the access method for a node is a format 4 DML statement, "FIND OWNER RECORD OF ... ", the loop in the procedure is degenerate. As each coset instance has only one owner record, there will be no looping generated in this case.

When looping does occur, the loop generated has the following form :

1) Find the first record using the access method determined in pass 1.

2) Repeatedly execute steps 3) - 6) while the value of the DML status (Codasyl 1971) remains zero.

3) GET the data items used in the targetlist and qualification from the record.

4) If the local condition for this loop is not satisfied, transfer to step 6.

5) If this is the innermost loop, then process the targetlist, or perform the aggregate function processing, otherwise fall through to the loop inside the current one.

6) Find the next record for the loop, using the access method determined in pass 1.

This nested structure suggests that each loop be generated with a recursive procedure, and this is in fact done.

Before step 1, the whole procedure for processing queries is recursively invoked for any subqueries which have not already been processed, and whose arguments are fully defined at that point. The subquery may be actually used inside a more deeply nested loop. A basic heuristic in code optimisation, however, is that loop invariant code should be moved outside loops where possible. Using this criterion to process the query has this effect. If a query has no arguments, as in 4.1.6, this heuristic will also have the effect of generating code to evaluate that query before evaluating any other query.

Two cases arise when a query to be processed has a V-graph containing nodes which are global to that query.

The first case is where only one node in the query is global, as depicted in Figure 7.15. This global node corresponds to a value which is fixed for a single evaluation of the query. No looping code will be generated for this node, which will be the start node of the query, by 8.2.1 .

The second case arises where more than one node in the V-graph is global. One of the global nodes will become the start node of the query. For the other global nodes, tests must be included in the code to reject a candidate record if its database key does not equal the relation variable database key for the global node. This global node database key will have been determined in a higher level query.

In both cases, the V-graph containing the global nodes may be regarded as having to be matched against the D-graph in all possible ways, with the V-graph nodes corresponding to global variables always matching previously fixed D-graph nodes.

8.3.2 Currency and UWA Usage

Generation of code for a particular loop is done without the procedure generating the code knowing anything about the other loops in the query, let alone loops in different queries. These other loops may affect processing in two ways. Firstly, an inner loop in this query or in another query might disturb currency information (Codasyl 1971) needed in finding subsequent records in the loop (Step 6 in 8.3.1). Secondly, if the range of the variable for such an inner loop is the same as the range of the current loop variable, the possibility exists that user working area locations stored into by the GET DML statement of the current loop, might be overwritten by the inner loop before any use could be made of them.

The currency information can be guaranteed correct by saving it before processing inner loops and restoring it afterwards. The saving may be done by storing the run-unit currency (Codasyl 1971) in a program variable (which has the same name as the relation variable for that loop). This must be done before step 5 of the loop procedure. The restoration may be done by including a format 1 FIND statement just before finding the next record in step 6. This would produce correct code, but would introduce an extra DML statement into each loop, which would often be unnecessary. Two steps may be taken to reduce the necessity for this format 1 FIND.

Firstly, at the cost of some slight complication, the loop generation process returns a list of the currencies changed by its loop and any inner loops, including those in other queries. If the format 1 FIND statement is not necessary to restore the currencies, it is not generated.

Secondly, a flag in the translator may be set to cause each FIND DML statement to suppress all coset and area currencies that are not going to be needed in following any path from the current node. This makes it less likely that any format 1 FIND statements will need to be generated.

It is possible to circumvent the problem of overwriting the user working area by following the GET DML statement of each loop by a set of statements to save those user working area locations just set by the GET statement. These save locations have names manufactured from the loop variable and the attribute name. In effect the user working area locations used in a particular loop are duplicated, and the duplicate locations used in subsequent tests and targetlist manipulations. The program is correct with these assignments included, although most of them are usually unnecessary. Methods for the removal of the unnecessary ones will be discussed in 8.5 .

8.4 Boolean test generation in the presence of subqueries

There is a lot of scope for optimisation in the translation of the tests on boolean expressions that are performed as part of the V-graph traversal. Tests should be translated firstly so that parts of the boolean expression that cannot affect the result are not evaluated, and secondly so that terms which are expensive to evaluate (containing subqueries) are not evaluated unless necessary.

8.4.1 Standard Boolean Test Generation

Begin by considering the translation of a boolean expression without subqueries. This problem has been addressed in (Arden 1962), (Bottenbruch 1962), and (Gries 1971), and indeed by numerous compiler writers. The following formulation is included not because it is original, but because to my knowledge it has not been published in this form before.

The procedure BTG (boolean test generation), has three parameters. The first is a boolean expression in the form of a tree whose internal nodes are the logical operators AND and OR, and whose leaves are terms. The second and third parameters are labels to be transferred to when the expression is true or false respectively. One of these label parameters may be the constant "FALLTHRU", indicating that control will fall through to the next statement in sequence rather than transferring to a label when the corresponding result of the expression is obtained.

It is assumed that, if the expression is not a term, the root of the expression tree is "AND" or "OR", and may be tested, and that each branch of the expression tree may be obtained and passed to a procedure. The procedure NEGATE inverts the expression tree using DeMorgan's laws, applied recursively. The parameterless function NEWLABEL generates a new label which may be used in the generated code. The syntax of the language used in describing this algorithm is straightforward, and similar to CODE-A.

```
PROCEDURE BTG(expression,truelabel,falselabel)
declare L1 as a variable of type label
IF expression is a term THEN
BEGINIF
    IF truelabel = "FALLTHRU" THEN
    BEGINIF
    expression <- NEGATE(expression)
    truelabel <-> falselabel < swap the labels >
    ENDIF
    GENERATE("IF" expression "THEN GOTO" truelabel);
    IF falselabel NE "FALLTHRU"
    THEN GENERATE("GOTO " falselabel);
ELSE
CASE root of expression OF
"AND" :
    IF falselabel = "FALLTHRU"
    THEN L1 <- NEWLABEL()
    ELSE L1 <- falselabel ;
    FOR B IN branches of expression tree DO
    BTG(NEGATE(B),L1,"FALLTHRU");
    IF truelabel NE "FALLTHRU"
    THEN GENERATE("GOTO" truelabel);
    IF falselabel = "FALLTHRU"
    THEN GENERATE(L1 ":");
"OR" :
    IF truelabel = "FALLTHRU"
    THEN L1 <- NEWLABEL()
    ELSE L1 <- truelabel ;
    FOR B IN branches of expression tree DO
    BTG(B,L1,"FALLTHRU");
    IF falselabel NE "FALLTHRU"
```

```
            THEN GENERATE("GOTO" falselabel);
            IF truelabel = "FALLTHRU"
            THEN GENERATE(L1 ":");
    ENDCASE
    ENDIF
```

One of the decisions made in designing the intermediate language for ALF was that control structures, including IF statements, should be nested in a last in first out fashion. This nesting requirement requires that the following code shall be produced for the conjunctive expression T1 AND T2 AND ... Tn.

```
IF T1 THEN
BEGINIF
IF T2 THEN
BEGINIF
:
IF Tn THEN
BEGINIF
:
code to be evaluated when expression is true
ENDIF
:
ENDIF
```

If the conditional statement containing the conjunctive expression contained an ELSE branch, that is, code to be executed when the conjunctive expression was false, then a supplementary boolean variable would have to be set to false before testing T1 and set to true inside the innermost IF test. This flag would then be tested after the ENDIF corresponding to the outermost IF loop.

The use of a supplementary boolean variable is also necessary for a nested testing of the disjunctive expression T1 OR T2 OR ... Tn.

```
B1 <- true
IF NOT T1 THEN
BEGINIF
:
IF NOT Tn THEN
BEGINIF
none of the Ti are true
B1 <- false
ENDIF
:
ENDIF
IF B1 = true THEN
BEGINIF
```

```
the disjunctive expression is true
ELSE
the disjunctive expression is false
ENDIF
```

The procedure to generate the boolean tests in this nested form follows the same general pattern as the previous algorithm, and it will not be repeated here.

The first sort of generated code contains control structure spaghetti in the form of labels and goto statements, whilst the structured code contains extraneous boolean variables and many levels of nesting. It is uncertain which is more difficult for a human to read.

It is worth noting that different code would be generated if the boolean expression tree was given in a structurally different, but logically equivalent form.

8.4.2 Introduction of Subqueries

In this section I shall be concerned with showing how one of the techniques for generating code to evaluate a boolean test, described in 8.4.1, may be applied in ALF to a boolean test containing subqueries. These techniques could be applied in any situation where there were differing evaluation costs for the primitive terms in the boolean expression.

The factorisation algorithm of Hall and Todd (Hall 1974), is extensively used in ALF. It factorises a boolean expression into two conjuncts, one containing only nominated terms. Boolean expressions in ALF are built up using terms and the logical operators AND and OR. This system of boolean expressions constitutes a boolean algebra (Gilbert 1976), and one may appeal to the duality properties of boolean algebras to transform the factorisation algorithm into a dual algorithm. This is done by exchanging occurrences of AND and OR, and occurrences of the constants TRUE and FALSE, in the original algorithm. The new algorithm, called *factorise-dual*, splits a boolean expression into two disjuncts, one containing only previously nominated terms. As in the original algorithm, the expression containing the nominated terms only is maximal.

In the evaluation process, a tree is generated called the *derived tree* of the boolean expression. In this tree each internal node is AND or OR, the branches under each node are ordered left to right, and a boolean expression is attached to each leaf node. The derived tree is equivalent to the original boolean expression, considering the leaf expression as being substituted for the leaf node. In addition, each leaf node is tagged with the queries which have to be processed before all the terms in the leaf node expression may be evaluated.

The ordering on the branches in the derived tree is such as to postpone evaluation of subqueries as long as possible when the derived tree is used as input to one of the code generation algorithms of 8.4.1.

1) All the queries in the boolean expression are ranked in estimated order of cost of evaluation. Currently this ranking is based only on the number of levels of subquery in each query, those containing fewest levels having the least cost. Other estimators, such as the size of the V-graph for each subquery, would give a more realistic estimate of the evaluation cost.

2) The terms containing queries for which code has not been generated are categorised as *hard*, and the rest are categorised as *easy*. Some terms containing queries may be easy at this stage, as the query may have been processed already either by being moved out of the current loop as described in 8.3.1, or by being amalgamated with another, already processed, query, as described in 7.7.

3) Try a conjunctive factorisation, to extract the maximal conjunct containing only easy terms. If this fails, try a disjunctive factorisation (*factorise-dual*), to extract the maximal disjunct containing only easy terms. If this fails also, the terms which would be able to be evaluated by processing the next easiest query are categorised as easy, and the factorisations are attempted again. If either of the factorisations succeeds, the boolean expression tree is split into a (conjunctive or disjunctive) factor and a residue. The factorisations are attempted again on the residue, using the dual type of factorisation first. Each time a factorisation is successful, a new part of the derived tree is created. The factor becomes a leaf of the derived tree, and is tagged with the expression it represents, and the queries which would have to be evaluated to evaluate all the terms in that expression. This process terminates, as, if the factorisation continually failed, all the subqueries in the expression would eventually be marked as processed, at which stage all the terms would be easy, and the factorisation would succeed (trivially).

4) The derived query tree is used as input to a code generation algorithm similar to those of 8.4.1, except that when a node for which queries have to be processed is encountered, the procedure to generate code to process the query is called before the test involving that query is generated.

As an example, consider the expression

T1(Q) AND (T2 OR T3).

The easy terms are T2 and T3. The hard term, which involves query Q, is T1. Conjunctive factorisation gives (T2 OR T3) as the factor, and T1(Q) as the residue. Both types of factorisation make no impression on the single hard term, so T1 is made easy by considering Q as processed. The derived tree is X1 AND X2, where (T2 OR T3) is attached to X1 and T1 is attached to X2. Also attached to X2 is the fact that Q must be processed before T1 could be evaluated. For this example, the following code would be generated. The code is in CODE-A.

```
IF T2 OR T3 THEN
BEGINIF
compute the query Q
IF T1 THEN
BEGINIF
    :
ENDIF
ENDIF
```

There is a choice as to whether the expressions on a leaf node of the derived tree have code generated for them at the most primitive level, that is allowing only terms, with no logical operators, to appear in IF statements, or whether composite boolean expressions are to be allowed, as in the above code fragment. As ALF is currently used to generate a higher level language, CODE-A, the latter course has been chosen for reasons of readability. If assembly language were being generated, it would probably be necessary to generate tests on component terms. This involves trivial changes to the test generation algorithm.

8.5 Code Optimisation

The intermediate language generated by ALF is fully described in Appendix B. It has been designed so that each statement can be considered to be free of side effects. The program variables which may be changed by intermediate language statements consist of user working area variables, the relation variables, the variables introduced by combining the relation variables with the names of the associated retrieved attributes, the special variables STATUS and CURRENT, the variables introduced as intermediate and result variables in aggregate function evaluation, and the currency status variables.

There is a currency status variable for each coset and for each area in the schema. Record currency (Codasyl 1971) plays no part in the intermediate language, and hence no variables for it are included.

Each DML FIND statement is translated into a number of assignment statements of the form

currency variable <- expression

The particular FIND command involved is treated as a side effect free operator in the same way as the arithmetic operators + and -. It has as operands the coset or area currency, or CURRENT, the run unit currency, and the area, coset, record and data item names concerned. There is one assignment statement generated for each currency variable altered by the FIND statement, and one for the variable STATUS, and the alteration of these variables is considered to be the only effect of the FIND statement. The currencies altered are those of all the cosets in which the record participates, and the area, excluding those whose updates have been inhibited with a SUPPRESS clause. For a discussion of where the SUPPRESS clause is used, see 8.3.2.

Similarly each DML GET statement appears as a number of assignment statements of the form:

user working area location <- expression

There is one such assignment statement for each data item retrieved with the GET statement. The *expression* uses GET as a side effect free operator, with CURRENT, the record name, and the data item name as operands.

Viewing each DML statement as a series of assignment statements allows standard compiler optimisation techniques to be used. Removing a single assignment statement by code optimisation modifies the DML statement, but the whole DML statement is not removed until all the assignments comprising it are removed.

The intermediate code output from pass 2 will usually contain some unnecessary assignment statements in addition to those which are components of DML statements. The reason for their presence is discussed fully in 8.3.2. Basically, to keep the complexity of the code generation passes to a manageable level, fail-safe code is generated in pass 2. This code may be improved at a later stage. The assignments which may not be necessary follow each DML GET statement, where the user working area items stored into by the GET are moved into another set of variables. There is one such set of variables for each loop containing a GET statement. These variables correspond to the items of the original query, and their contents are used in subsequent tests and in the targetlist. If the user working area variable from which a save variable is derived is not subsequently changed by another GET statement, then the user working area variable may itself be used in subsequent tests and in the targetlist, and the assignment statement eliminated. This may be done using standard methods for the optimisation of algebraic languages, for example those described in Chapter 6 of (Cocke 1970), or in Vol II of (Aho 1973).

The intermediate language generated by ALF has been structured to assist this optimisation pass. Simpler analysis than that given in the above references is possible when all control structures are fully nested, as in the intermediate language of ALF. Whilst the generated programs would be more aesthetically pleasing if the unnecessary assignments were removed, the improvement in execution performance would probably not be great. The methods used in (Zelkowitz 1973), and in (Hecht 1977) are applicable here.

This optimisation pass has not been implemented in the current version of ALF.

currency, or CURRENT, the run unit currency, and the area, coset, record and data item names concerned. There is one assignment statement generated for each currency variable altered by the FIND statement, and one for the variable STATUS, and the alteration of these variables is considered to be the only effect of the FIND statement. The currencies altered are those of all the cosets in which the record participates, and the area, excluding those whose updates have been inhibited with a SUPPRESS clause. For a discussion of where the SUPPRESS clause is used, see 8.3.2.

Similarly each DML GET statement appears as a number of assignment statements of the form:

user working area location <- *expression*

There is one such assignment statement for each data item retrieved with the GET statement. The expression uses GET as a side effect free operator, with CURRENT, the record name, and the data item name as operands.

Viewing each DML statement as a series of assignment statements allows standard compiler optimisation techniques to be used. Removing a single assignment statement by code optimisation modifies the DML statement, but the whole DML statement is not removed until all the assignments comprising it are removed.

The intermediate code output from pass 2 will usually contain some unnecessary assignment statements in addition to those which are components of DML statements. The reason for their presence is discussed fully in 8.3.2. Basically, to keep the complexity of the code generation passes to a manageable level, fail-safe code is generated in pass 2. This code may be improved at a later stage. The assignments which may not be necessary follow each DML GET statement, where the user working area items stored into by the GET are moved into another set of variables. There is one such set of variables for each loop containing a GET statement. These variables correspond to the names of the original query, and their contents are used in subsequent tests and in the targetlist. If the user working area variable from which a new variable is derived is not subsequently changed by another GET statement, then the user working area variable may itself be used in subsequent tests and in the targetlist, and the assignment statement eliminated. This may be done using standard methods for the optimisation of algebraic languages, for example those described in Chapter 6 of (Cocke 1970), or in Vol II of (Aho 1973).

The intermediate language generated by ALP has been structured to assist this optimisation pass. Simpler analysis than that given in the above references is possible when all control structures are fully nested, as in the intermediate language of ALP. While the generated programs would be more aesthetically pleasing if the unnecessary assignments were removed, the improvement in execution performance would probably not be great. The methods used in (Zelkowitz 1975), and in (Hecht 1977) are applicable here.

9.0 Extensions and Problems

In this section I will consider some possible extensions and improvements to the techniques described in this paper.

9.1 Updates

No update commands are currently included in ALF, although their inclusion would not be difficult. An update command would have to have at least the ability to create a tuple in a relation, and the ability for changing attribute values in an already existing tuple, or set of tuples.

In creating a tuple, values would have to be specified at least for the primary key attributes. For the relations implemented as records which are members of non SYSTEM owned cosets, some of these primary key attributes might be virtual, coset defining attributes. Values of these attributes specify coset occurrences into which the record representing the tuple must be stored. The most convenient way to do this is to use the *set occurrence selection* facilities in the DDL (Codasyl 1971). These facilities allow selection of a coset occurrence by setting identifying user working area data items along a path between the coset owner and a root record. The values of these data items allow the system to determine particular record, and hence coset occurrences along the path.

Consider the schema of fig 4.1, and assume that an EMP tuple is being stored. To do this, values must be supplied for the virtual attributes C# and D# in EMP, as well as the other attribute in the primary key, E#. If the DDL schema specification contained a declaration of the form

SET OCCURRENCE SELECTION IS THRU
LOCATION MODE OF OWNER USING D#

for the coset with EMP as member, and a declaration of the form

SET OCCURRENCE SELECTION IS THRU
LOCATION MODE OF OWNER USING C#

for the coset with DEPT as member, then the (actual) user working area items DEPT.D# and COMP.C# could be set to the supplied values, and a STORE DML command issued. If no COMP tuple contained the supplied value of C#, or if no DEPT tuple in the coset defined by that COMP tuple contained the supplied value of D#, then the STORE command would not succeed. This rejection implements an important integrity constraint.

In altering an already existing tuple, two cases arise.

The first is where primary key attributes are altered, if this is to be allowed at all. In this case, the same strategy as before could be used. The record to be altered would be found, the new values of the key attributes set in the user working area, and a DML MODIFY command (Codasyl 1971), issued. This would move the record into different coset occurrences, reflecting the altered values of the virtual attributes. Errors would occur, as before, if owner records containing the modified values were not present in the database.

The second case is where non primary key attributes are altered. This is done in a straightforward way, by finding the record to be modified, setting the user working area locations corresponding to the attributes to be modified, and issuing a DML MODIFY command.

This allows the new values in a tuple to be functions of the previous values in the tuple. Thus one could say, in a notation similar to INGRES (Stonebraker 1976),

```
REPLACE EMP.SAL = EMP.SAL + 500
WHERE EMP.E# = 123 .
```

to mean "Increase the salary of the employee whose number is 123 by $500".

Updates could also be performed on all members of a set of tuples, for example, "Increase the salaries of all employees in department ABC of company XYZ, who are below grade 2, by ten percent".

```
REPLACE EMP.SAL = 1.1*EMP.SAL
WHERE
EMP.D# = "ABC"
AND EMP.C# = "XYZ"
AND EMP.GRADE LT 2 .
```

A different strategy could be adopted when storing a new tuple, or modifying the primary key attributes of an existing tuple, where some of the primary key attributes are virtual. Whenever a virtual attribute value could not be materialised, due to the absence of the record containing the corresponding actual attribute, new owner records containing the actual attributes could be stored and linked into the appropriate coset occurrences.

Tuple deletion falls into two classes. The first is where the primary key of the relation from which the tuple is to be deleted is not a foreign key in any other relation; that is, in the network model, the record is not the owner of any cosets. In this case, the record may be deleted, and integrity will be preserved.

The second case occurs when the primary key of the tuple to be deleted is a foreign key in another relation. In the network model, this means that the record instance being deleted is the owner of a coset. Several courses of action may be taken.

1) The deletion may cause an error if any of the cosets that the record owns is non-empty.

2) All the coset members may be deleted.

3) The record may be deleted, and all the members unlinked from each coset that the record owns. For this to be possible, membership of each coset involved would have to be OPTIONAL.

All the machinery so far developed for selecting records, including aggregate functions, could be used in implementing update commands similar to those sketched out here.

9.2 Security and Integrity

The approach to security taken in (Stonebraker 1974b) and (Stonebraker 1975) is to conjoin a condition to the qualification part of each retrieval or update statement. There may be one such condition for each user, and one for each type of access (update or retrieval). No measures of this sort are currently included in ALF. The condition which is added to the qualification does not cause any rejection visible to the user, but merely makes tuples not satisfying the condition invisible to the user. This approach has the very obvious advantages of being easy to implement, as it uses the already exising machinery for handling qualifications on commands, and in incurring a very low overhead. Whether it solves all integrity problems for all applications is not clear, (for example it may be desirable to report on attempted security violations), nevertheless the other security provisions provided by Codasyl systems are also available, provided methods of using them can be built into higher level, ALF like languages.

9.3 More General Joins

The ALF implementation concentrates on the efficient processing of joins in which a coset may be used, that is, joins in which a candidate key of one relation is specified as equal to the corresponding foreign key in another. Some different types of join are handled efficiently, others are not.

An example of a non-coset join is where the primary key in one relation is specified equal to an expression involving attributes of other relations. In this case, ALF would generate code to find the tuples in the other relations first, compute the primary key using attribute values from those other tuples, and use the key to find a admissable tuple in the first relation.

Joins which are not equi-joins are not processed efficiently. For each tuple of the first relation in the join, all tuples in the second relation will be scanned, and the join condition tested. The number of tuples scanned may be reduced using other conjuncts in the qualification.

Equi-joins which do not specify candidate keys and the corresponding foreign keys must usually be processed by the brute force approach used for non equi-joins. However, if it were considered justified by the Database Administrator, extra structures could be included in the network schema to assist in processing this sort of join.

Consider two different relations containing a similar non-key attribute, say *colour*. The processing of an equi-join on that attribute would be improved if an extra record and two extra cosets were defined. The extra, introduced record would be the owner of both cosets, and the records containing the non-key attribute would each be a member of one of the cosets. Each coset instance would group all tuples in one of the records with identical values of the non-key attribute. For example, all the *green* records of one type would be grouped by one of the cosets under one owner instance, and all the *green* records of the other type would be grouped under the same owner instance, in the other coset. Call such a structure a *coupling*. There is no reason why more than two records could not participate in a coupling, nor is there any reason why there should not be more than one attribute in each relation involved.

If an equi-join term using the coupling attributes could be factored out of the qualification of the ALF statement, then an extra node, ranging over the coupling record, and two arcs representing each of the coupling cosets could be added to the V-graph of the query. This would cause the coupling to be exploited when evaluating the join.

9.4 Views

A *view* is a way of modifying the way that a user sees the data in a Database. In this sense, the mapping used in this paper defines a relational view of the network data. Furthur levels of view could be defined on top of this relational view in the following way.

1) A restriction could be applied to the tuples of a single relation.

2) A composite relation, made up from several other relations using joins of various sorts could be defined.

3) Steps 1) and 2) could be combined.

A mechanism for defining and using views has been specified and implemented in ALF. Each view may be thought of as a virtual relation, and may be used in the same way as an actual relation.

As an example, consider the schema of Figure 4.1, and assume that a view relation, called PD, is to be defined using the base relations PROJECT, DEPT, and PROJDEPT. Furthur, assume that the view relation is to have attributes D#, C#, PROJ#, DBUDGET and PBUDGET, the last two being the department and project budgets respectively. The user of the view is not to see departments outside the ACT, or projects with budgets greater than 1000000. Using the view syntax implemented in ALF, the view would be defined as follows:

```
DEFINE VIEW PD
FROM D RANGE DEPT, P RANGE PROJECT, RR RANGE
PROJDEPT
USING D.D#, D.C#, P.PROJ#, D.BUDGET RENAMED
DBUDGET, P.BUDGET RENAMED PBUDGET
WHERE D.D#=RR.D# AND D.C#=RR.C# AND
P.PROJ#=RR.PROJ# AND P.BUDGET LT 1000000 AND
D.DLOC="ACT".
```

The user could then declare relation variables with range PD and retrieve from the view as if it were a single relation. A user need not know anything about joins in accessing the data, nor need he know anything about tuples that have been masked out of the view.

The problems involved in updating views have been touched on in 9.1. More work needs to be done to define update operations on views, in those cases where updates are possible.

The following steps are taken in implementing views in ALF. For each variable in an ALF statement which ranged over a view, new relation variables corresponding to the dummy variables specified in the FROM clause of the view definition are created. These are substituted for the dummy variables in the qualification in the WHERE clause of the view definition. Thus each instance of the view has it's own set of variables, and the user may reuse those variables used in defining the view.

This transformed qualification is conjoined to the ALF statement qualification, and each item using a view variable is changed to it's source item from the view definition. The ALF statement is then processed as before.

This method of view implementation is similar to macro substitution prior to translation, and does not affect the translation process itself in any way.

Views may be defined using other views as well as base relations, and may contain subqueries in the qualification of the view. Some view binding time convention must be adopted; that is a choice has to be made between binding the definition of a view defined in terms of other views at definition time or binding at usage time. At present, binding is done at usage time, but a separate command allows the user to define views in terms of actual relations. Use of this command allows redefinition of views without affecting other views defined in terms of the redefined view.

10. Conclusion

the ALF translator has been implemented as an interactive system running on the CSIRO CYBER 76 computer. It has been implemented to demonstrate the feasibility of compiling efficient object code from higher level query commands.

ALF approaches Relational Database Implementation from a different direction to other systems implemented so far. Building a Relational System on top of an underlying network system allows a relational interface to be implemented with a fraction of the effort needed to build such a system from the ground up.

The mapping between network and relational schemata, and the translation techniques employed, allow programs to be generated which exploit the available access methods at least as efficiently as the average programmer.

The translation model is sufficiently flexible to allow extensions such as updates, integrity and security measures, joins not defined by cosets, and views, to be implemented in a straightforward manner.

Bibliography and References

The references which follow are arranged in alphabetical order of first author, and are referred to in the text in the form (Author year). If two or more references by a particular first author are published in one year, a lower case alphabetic index is used to differentiate them. Not all the entries in this Bibliography are referenced in the text; some have been included because of their relevance to the subject matter of this report.

In this bibliography, JACM refers to the Journal of the ACM, CACM to the Communications of the ACM, TODS to the ACM Transactions on Database Systems, and UKSC to publications of the IBM United Kingdom Scientific Centre, Peterlee, Durham.

(Aho 1972)
Aho, A. V. and Ullman, J. D.
"The Theory of Parsing, Translation, and Compiling"
Vol I, Parsing, Vol II, Compiling.
Prentice-Hall, 1972

(ANSI 1975)
"ANSI/X3/SPARC Study Group on Data Base Management Systems - Interim Report"
SIGMOD FDT 7 2 1975

(Arden 1962)
Arden, B. W., Galler, B. A. and Graham, R. M.
"An Algorithm for translating Boolean Expressions"
JACM Vol 9 (1962) pp222-239

(Astrahan 1975)
Astrahan, M. M. and Chamberlin, D. D.
"Implementation of a Structured English Query Language" CACM 18 10, (Oct 1975) pp580-588

(Astrahan 1976)
Astrahan, M. M. et al
"System R : A Relational Approach to Database Management".
TODS Vol 1 No 2 (June 1976)

(Bachman 1969)
Bachman, C. W.
"Data Structure Diagrams"
SIGBDP : Database 1 2 (1969)

(Bobrow 1974)
Bobrow, D. G. and Raphael, B.
"New Programming Languages for Artificial Intelligence Research"
ACM Computing Surveys 6 3 (Sept 1974) pp 155-174

(Bottenbruch 1962)
Bottenbruch, H. H. and Grau, A. A.
"On Translation of Boolean Expressions"
CACM Vol 5 (1962)

(Boyce 1974)
Boyce, R. F., Chamberlin, D. D., King, W. F. and Hammer, M. M.
"Specifyimg Queries as Relational Expressions"
in (Klimbie 1974) pp169-178

(Chamberlin 1974)
Chamberlin, D. D. and Boyce, R. F.
"SEQUEL, A Structured English Query Language"
Proc. ACM SIGFIDET Workshop on Data Description, Access, and Control, May 1974.

(Chamberlin 1976)
"Relational Data Base Management Systems"
in (Sibley 1976)

(Cocke 1970)
Cocke, J. and Schwartz, J.
"Programming Languages and their Compilers"
Lecture notes, State University of New York, 1970.

(Codasyl 1971)
Codasyl Data Base Task Group
"April 1971 Report"
Available from ACM.

(Codd 1970)
Codd, E. F.
"A Relational Model for Large Shared Data Banks"
CACM 13 6 (June 1970)

(Codd 1971a)
Codd, E. F.
"Normalised Data Base Structure - A brief Tutorial"
Proc 1971 ACM SIGFIDET Workshop.

(Codd 1971b)
Codd, E. F.

"A Database Sublanguage founded on the Relational Calculus"
Proc 1971 ACM SIGFIDET Workshop.

(Codd 1972a)
Codd, E. F.
"Furthur Normalisation of the Data Base Relational Model"
In (Rustin 1972)

(Codd 1972b)
Codd, E. F.
"Relational Completeness of Data Base Sublanguages"
in (Rustin 1972)

(Codd 1975a)
Codd, E. F.
"Implementation of Relational Database Management Systems"
ACM SIGMOD Bulletin 7, 3 and 4, 1975.

(Codd 1975b)
Codd, E. F.
"Understanding Relations" ACM SIGMOD Bulletin 7, Nos 1 to 4, 1975

(Codd 1979)
Codd,E. F.
"Extending the Data Base Relational Model to Capture More Meaning"
Proc. Australian Computer Science Conference, Hobart, Feb 1-2 1979.

(Date 1975)
Date, C. J.
"An Introduction to Database Systems"
Reading: Addison Wesley, 1975

(Douque 1975)
Douque, B. C. M. and Nijssen, G. M. (eds)
"Data Base Description"
IFIP Special Working Conference on Data Description languages : An In-depth Technical Evaluation of the Codasyl DDL. 13-17 Jan 1975
North-Holland 1975

(Engleman 1975)
Engleman, C.
"Engineering of Quality Software Systems - Towards an Analysis of the LISP Programming Language"
Mitre Corp. Jan 1975 AD-A007 769

(Friedman 1969)
Friedman, D. P., Dickson, D. C., Fraser, J. J., and Pratt, T. W.

"GRASPE 1.5 A Graph Processor and its Application"
University of Houston, 1969

(Gilbert 1976)
Gilbert, W. J.
"Modern Algebra with Applications"
Wiley 1976

(Gimpel 1973)
Gimpel, J. F.
"A Theory of Discrete Patterns and their Implementation in SNOBOL4"
CACM 16 2 (Feb 1973) pp 91-100

(Gries 1971)
Gries, D.
"Compiler Construction for Digital Computers"
Wiley 1971

(Gudes 1973)
Gudes, E. and Reiter, A.
"On Evaluating Boolean Expressions"
Software Practice and Experience, Vol 3 (1973) pp 345-350

(Hall 1974a)
Hall, P. A. V. and Todd, S. J. P.
"Factorisation of Algebraic Expressions"
IBM UKSC Report UKSC 0055, April 1974.

(Hall 1974b)
"Common Subexpression Identification in General Algebraic Systems"
IBM UKSC Report UKSC 0074, November 1974.
Hall, P. A. V. and Todd, S. J. P.

(Hall 1974c)
"User Functions and Data Bases"
IBM UKSC TN 01, May 1974

(Hall 1975a)
Hall, P. A. V.
"Optimisation of a Single Relational Expression in a Relational Database System"
IBM UKSC Report UKSC 0076, June 1975.

(Hall 1975b)
Hall, P. A. V. and Todd, S. J. P.
"Database Administrator Facilities in PRTV"
IBM UKSC TN 22

(Hecht 1977)
Hecht, M.
"Flow Analysis of Computer Programs"
North Holland 1977

(Hewitt 1972)
Hewitt, C.
"Description and Theoretical Analysis of PLANNER"
MIT AI TR-258 (April 1972)

(Hitchcock 1975)
Hitchcock, P.
"User Extensions to the Peterlee Relational Test Vehicle"
IBM UKSC TN 33 December 1975

(Kalinichenko 1976)
Kalinichenko, L. A.
"Relational-Network Data Structure Mapping"
in (Nijssen 1976) pp 303-310

(Kay 1975)
Kay, M. H.
"An Assessment of the Codasyl DDL for use with a Relational Subschema"
in (Douque 1975) pp199-214

(Kerr 1975)
Kerr, D. S. (ed)
"Very Large Data Bases"
Proc. International Conference on Very Large Data Bases, Sept 1975
Available from ACM

(Klimbie 1974)
Klimbie, J. W. and Koffman, K. L. (eds)
"Data Base Management"
Proceedings of the IFIP working conference on Database Management held at Cargese, Corsica, 1-5 April 1974
North-Holland 1974

(Lorie 1979)
Lorie, R. A. and Nilsson, J. F.
"An Access Specification Language for a Relational Data Base System"
IBM Journal of Res. and Develop., 23 3 (May 1979), p286

(Mackenzie 1974)
Mackenzie, H. G. and Smith, J. L.
"Fordata Reference Manual"

CSIRO Division of Computing Research (August 1974) (Revised 1977).

(Mackenzie 1977a)
Mackenzie, H. G. and Smith, J. L.
"The Implementation of a Database Management System"
Australian Comp. J. 9 4 (Nov 1977)

(Mackenzie 1977b)
Mackenzie, H. G.
"Codasyl Database Management Systems"
in "Database Management Systems", ed. Wolfendale, G. L.
ANU Press, Canberra 1977

(Mackenzie 1977c)
Mackenzie, H.G. and Kelly, G.
"A Query/Update Package for Library or Personal Reference Use"
Australian Computer Journal, November 1977, (pp 155-158)

(McGee 1974)
McGee, W. M.
"A Contribution to the Study of Data Equivalence"
in (Klimbie 1974) pp123-148

(Martin 1975)
Martin, J.
"Computer Data Base Organisation"
Prentice Hall 1975

(Metaxides 1975)
Metaxides, A.
"Information Bearing and Non Information Bearing Sets" in (Douque 1975) pp363-368

(Miller 1968)
Miller, W. F. and Shaw, A. C.
"Linguistic Methods in Picture Processing : - A Survey"
Proc AFIPS FJCC 33,
Thompson, Washington pp279-290

(Nijssen 1974)
Nijssen, G. M.
"Data Structuring in the DDL and Relational Data Model"
in (Klimbie 1974)

(Nijssen 1975)
Nijssen, G. M.
"Set and Codasyl Set or Coset"

in (Douque 1975) pp1-72

(Nijssen 1976)
Nijssen, G. M. (ed)
"Modelling in Data Base Management Systems"
Proc. IFIP Working Conference on Modelling in Data Base Management Systems, Freudenstadt, Germany, 5-8 Jan, 1976

(Olle 1975)
Olle, T. W.
"A Practitioners view of Relational Data Base Theory"
SIGFDT Bulletin Vol 7 Nos 3 and 4, 1975

(Owlett 1976)
Owlett, J.
"Deferring and Defining in Databases"
IBM UKSC TN 35, Revised Nov 1976

(Palermo 1974)
Palermo, F. P.
"A Data Base Search Problem"
in "Information Systems - COINS IV", ed Tou, J., Plenum 1974

(Pecherer 1976)
Pecherer, R. M.
"Efficient Exploration of Product Spaces"
Proc. 1976 SIGMOD Conference on Management of Data

(Rothnie 1975)
Rothnie, J. B. Jr.
"Evaluating Inter-Entry Retrieval Expressions in a Relational Data Base Management System"
Proc 1975 NCC, pp417-422

(Rulifson 1972)
Rulifson, J. F., Derksen, J. A. and Waldinger, R. J.
"QA4: A Procedural Calculus for Intuitive Reasoning"
Stanford AI Centre Technical Note 73 (November 1972)

(Rustin 1972)
Rustin, R. (ed)
"Data Base Systems"
Courant Computer Science Symposium 6
Prentice-Hall 1972

(Rustin 1974)
Rustin, R. (ed)

"ACM SIGMOD Workshop on Data Description, Access and Control" May 1-3 1974 Vols I and II.

(Sandewall 1978)
Sandewall, E.
"Programming in an interactive environment : The LISP Experience"
Computing Surveys 10 1 (March 1978)

(Sibley 1974)
Sibley, E. H.
"On the equivalence of Data Base Systems"
in (Rustin 1974) Vol II.

(Sibley 1976)
Sibley, E. H. (ed)
"Data Base Management Systems"
Special Issue, TODS 8 1 (March 1976)

(Siklossy 1975)
Siklossy, L.
"Let's Talk LISP"
Prentice-Hall, Englewood Cliffs, N.J. 1975

(Smith 1975)
Smith, J. M. and Yen-Tan Chang, P.
"Optimising the Performance of a Relational Database Interface"
CACM 18 10, (Oct 1975) pp568-579

(Stonebraker 1974a)
Stonebraker, M.
"A Functional View of Data Independence" Proc 1974 ACM SIGFIDET Workshop on Data Access and Control, May 1974

(Stonebraker 1974b)
Stonebraker, M. and Wong, E.
"Access Control in a Relational Data Base System by Query Modification" Proc 1974 ACM National Conference, Nov 1974

(Stonebraker 1975)
Stonebraker, M.
Implementation of Views and Integrity Constraints in Relational Data Base Systems by Query Modification" Proc 1975 SIGMOD Workshop on Management of Data, May 1975

(Stonebraker 1976)
Stonebraker, M. et al
"The design and implementation of INGRES"

TODS 1 3 (Sept 1976)

(Symonds 1970)
Symonds, A.J. and Lorie, R. A.
"A Scheme for describing a Relational Data Base"
Proc. ACM SIGFIDET Workshop on Data Description and Access, November, 1970

(Taylor 1976)
Taylor, R. W. and Frank, R. L.
"Codasyl Database Management Systems"
in (Sibley 1976)

(Texas 1975)
University of Texas Computation Centre
"LISP Reference Manual - CDC 6000"
University of Texas, Dec 1975

(Thomas 1975)
Thomas, J. C. and Gold, J. D.
"A Psychological Study of Query-by-Example"
Proc. AFIPS NCC, Vol 44, 1975, p 439

(Todd 1974)
Todd, S. J. P.
"Implementation of Join Operator in Relational Data Bases"
IBM UKSC TN 15, November 1974

(Todd 1975a)
Todd, S. J. P.
"PRTV, An Efficient Implementation for Large Relational Data Bases"
in (Kerr 1975) p 554

(Todd 1975b)
Todd, S. J. P.
"PRTV A Technical Overview"
IBM UKSC 0075, May 1975

(Todd 1976a)
Todd, S. J. P.
"The Peterlee Relational Test Vehicle - A System Overview"
IBM Systems Journal, 4 1976 pp285-308

(Todd 1976b)
Todd, S. J. P.
"Integrated Architecture for Transaction Specification and Optimisation in Relational Data Base Systems"

IBM UKSC Report UKSC 0085, November 1976

(Todd 1976c)
Todd, S. J. P.
"Automatic Constraint Maintenance and Updating Defined Relations"
IBM UKSC TN 44, November 1976

(Tsichritzis 1975)
Tsichritzis, D.
"A Network Framework for Relation Implementation"
in (Douque 1975) pp269-282

(Verhofstad 1976)
Verhofstad, J. S. M.
"The PRTV Optimiser : The Current State"
IBM UKSC Report UKSC 0083, May 1976

(Waite 1973)
Waite, W.M.
"Implementing Software for Non-Numeric Applications"
Prentice-Hall 1973

(Weissman 1967)
Weissman, C.
"LISP 1.5 Primer"
Dickenson, Belmont, Calif. 1967

(Wong 1976)
Wong, E. and Yousefi, K.
"Decomposition - A Strategy for Query Processing"
TODS 1 3 (Sept 1976)

(Yao 1979)
Yao, S. B.
Optimisation of Query Evaluation Algorithms"
TODS 4 3, (June 1979) pp133-155

(Zelkowitz 1973)
Zelkowitz, M. V., and Bail, W. G.
"Optimisation of Structured Programs"
Software Practice and Experience, Vol 4 No 1, Jan-March 1974

(Zimmerman 1975)
Zimmerman, K.
"Different Views of a Data Base: Coexistence Between Network Model and Relational Model" in (Kerr 1975) p 535

(Zloof 1975)
Zloof, M. M.
"Query-by-Example"
Proc. AfIPS NCC, Vol 44, 1975, p 431.

(Zloof 1976)
Zloof, M. M.
"Query-by-Example : Operations on the Transitive closure"
RC 5526 (Revised)
IBM T.J. Watson Research Centre, Yorktown Heights, N.Y.

(Zloof 1977)
Zloof, M. M.
"Query-by-Example : A Data Base Language"
IBM Systems Journal 16 4 (1977)

IBM UKSC Report UKSC 0085, November 1976

(Todd 1976a)
Todd, S. J. P.
"Automatic Constraint Maintenance and Updating Defined Relations"
IBM UKSC TN 44, November 1976

(Tsichritzis 1975)
Tsichritzis, D.
"A Network Framework for Relation Implementation"
in (Douque 1976) pp269-282

(Verhofstad 1976)
Verhofstad, J. S. M.
"The PRTV Optimiser: The Current State"
IBM UKSC Report UKSC 0083, May 1976

(Waite 1973)
Waite, W.M.
"Implementing Software for Non-Numeric Applications"
Prentice Hall 1973

(Weissman 1967)
Weissman, C.
"LISP 1.5 Primer"
Dickenson, Belmont, Calif. 1967

(Wong 1976)
Wong, E. and Youssefi, K.
"Decomposition - A Strategy for Query Processing"
TODS 1 3 (Sept 1976)

(Yao 1979)
Yao, S. B.
Optimisation of Query Evaluation Algorithms"
TODS 4 2, (June 1979) pp133-155

(Zelkowitz 1974)
Zelkowitz, M. V., and Bail, W. G.
"Optimisation of Structured Programs"
Software Practice and Experience, Vol 4 No 1, Jan-March 1974

(Zimmerman 1975)
Zimmerman, K.
"Different Views of a Data Base: Coexistence Between Network Model and Relational Model" in (Kerr 1975) p.288

Appendix A - Implementation Language

ALF is implemented using an extended version of UT LISP, which runs on CDC CYBER computers. UT LISP is documented in (Texas 1975). The extensions are a group of macros which enable some constructs similar to those found in other modern higher level languages to be used. LISP programmers commonly spend quite a bit of time making sure that the many parenthesis required by the language are correctly nested; the macro constructs described here reduce the number of levels of nesting as well as providing program text which is more natural for the programmer not raised on LISP to read.

The reasons for choosing LISP to implement this prototype system are many. For a general overview of them, the reader should consult (Engleman 1975). There were not, in fact, many alternatives, however even if the choice had not been constrained by what was available on the CYBER 76, the only other candidates that I would have considered suitable would have been more modern versions of LISP. Features which influenced the selection of LISP are given below.

1) Ease of representation and manipulation of symbolic data objects, the representation usually being changeable with only local effects.

2) Recursion

3) Automatic storage management

4) The existence of a graph processing package, GRASPE (Friedman 1974)

5) Good diagnostic and debugging facilities.

6) Uniform treatment of procedure and data.
This has two important consequences. The first is that a rigorous definition of the language may be formulated, and the second is that procedures may be synthesised and then executed using the standard evaluation mechanism.

7) The availability of an interactive envoronment for program development, in which programs and data can be defined, examined and altered on line, during execution.

The rest of this Appendix describes the language extensions. In describing the following language extensions, I have assumed a knowledge of LISP on the part of the reader; there are numerous introductory texts available, for example (Siklossy 1975), (Weissman 1967).

The extensions are :

1. IF Statement

(IF *expression* **THEN** *expression-list*

ELSEIF *expression* **THEN** *expression-list*

: : : :

ELSE *expression-list* **)**

This statement is expanded into a "COND" The expression following the IF is evaluated, and if it is true (that is, not NIL), each expression in the *expression-list* following the first THEN will be evaluated. There may be any number of ELSEIF clauses, and an optional final ELSE clause. If the first expression is false (that is, NIL) then the expression following the next ELSEIF is evaluated, and if true the expressions in the corresponding expression list are evaluated. If the expression is false, the process continues. If no expression is true, the expressions in the expression list following the final THEN are evaluated. The value of the whole statement is the value of the last expression evaluated.

2. FOR Statement

(FOR *atom* **IN** *list* **DO** *expression list* **)**

(FOR *atom* **IN** *list* **DO** *function* **)**

This statement is expanded into a MAPCAR. In the first version, the elements of *list* are successively bound to *atom* (that is *atom* successively takes the values of the elements of *list*) and the expression in expression list evaluated in the presence (environment) of this binding.

In the second version, *function* may be an atom which is the name of a function, (that is, an atom with an appropriate EXPR or FEXPR property) or a lambda expression. *function* must be a function of one argument. The effect is the same as in the first version, the function being applied to successive elements of *list*.

The value of this statement is a list comprising the values of each of the results of the function application.

3. CASE Statement

```
(CASE expression OF
label-list : expression-list ;
   :       :      :         :
ELSE expression list)
```

This statement expands into a SELECT. The expression following CASE is evaluated, and its value compared with each of the labels in the first *label-list* which of course may contain only one label. *label list* is merely a list of expressions, so this statement is considerably more general than the case statements found in other programming languages. When a match is found, the expressions in the expression list to the right of the colon are evaluated. If no match is found in any of the *label lists*, the expressions following the ELSE are evaluated.

The value of this statement is the value of the last expression executed. If there is no match on any of the expressions in any of the *label-lists* and there is no ELSE clause, then the value is NIL.

4. WHILE Statement

```
(WHILE expression DO expression list)
```

If *expression* evaluates to true (not NIL), then the expressions in *expression list* are evaluated. This process then repeats itself until *expression* becomes false.

5. REPEAT Statement

```
(REPEAT expression list UNTIL expression)
```

The expressions in *expression list* are evaluated, and then *expression* is evaluated. If *expression* is false the process repeats itself. This is similar to the WHILE statement, except that the test is performed at the end of the loop, and control stays in the loop if the test expression is false, not true as with WHILE.

6. ALL Statement

(ALL *atom* IN *list* SATISFIES *function*)

(ALL *atom* IN *list* SATISFIES *expression-list*)

As in the FOR statement, *function* is an atom which has been defined as a function, or a lambda expression. *function* is successively applied to each element of list. The value of the expression is false if one of these applications returns false, and succeeding ones are not done. If they all return true, the value is the value of the last one.

In the second version, successive elements of *list* are bound to *atom* and the expressions in expression list evaluated.

7. SOME Statement

(SOME *atom* IN *list* SATISFIES *function*)

(SOME *atom* IN *list* SATISFIES *expression-list*)

This statement is very similar to the ALL statement except that if any of the function applications returns true, then that is the value of the statement and no further function applications are done. If they are all false, the statement value is false.

8. NO Statement

(NO *atom* IN *list* SATISFIES *function*)

(NO *atom* IN *list* SATISFIES *expression-list*)

This statement is true if none of the list elements satisfies the function or expression list, in the same sense as in 6. or 7. It is false if any list element does, and no furthur evaluation is done.

Appendix B - The Intermediate language

The intermediate language generated by the ALF transator is designed to be translatable without undue difficulty into other high level languages, and also to be efficiently interpretable. It is represented as a list of quadruples, or *quads*, in the form

(*operator result operands* DML-atom)

The language does not contain labels or a GOTO statement, and all control structures are nested in a last in first out fashion. This was not done for aesthetic reasons, but to make it possible to implement the code optimisation and interpretation procedures in a recursive manner.

For code optimisation purposes again, each DML statement is viewed as being equivalent to a number of assignment statements. These assignment statements are grouped together by having the same atomic list item *DML-atom*, which has properties describing the DML statement.

The different non-DML operators which can occur are

1. ASSIGN
The value of the expression in the *operands* position is assigned to the variable in *result* .

2. WHILE
result is a boolean expression in prefix form; *operands* is an atomic identifier for the statement. The quads between the WHILE and the corresponding ENDWHILE quad are obeyed until the WHILE condition (*result*) becomes false.

3. ENDWHILE
Terminates the scope of a WHILE loop, *result* is the atomic identifier for the WHILE loop.

4. IF
result is a boolean expression, *operands* an atomic identifier for the IF statement. If the boolean expression is true, the following code down to a corresponding ENDIF or ELSE is executed. If false, then the code between the corresponding ELSE and the ENDIF is executed if there is an ELSE, otherwise control is transferred to the statement following the ENDIF.

5. ENDIF, ELSE
result is an atomic identifier corresponding to the same identifier in the

corresponding previous IF quad.

6. COMMENT
result contains commenting text, and the effect is purely documentary.

7. EXITWHILE
result is the identifier for a WHILE quad. This quad indicates that control should be transferred to the statement following the corresponding ENDWHILE statement.

If the atom DML-atom is non-NIL, then the quad represents one of the assignment statements comprising a DML statement. DML-atom has properties which enable the DML statement to be generated (or interpreted).

These properties are

1. DMLVERB
The DML verb in the DML statement. It may be one of GET, FIND1, FIND3, FIND4, FIND5, FIND6, FIND7, indicating the GET statement and the various forms of the FIND statement. (Codasyl 1971)

2. SETAR
The coset or area name involved.

3. DILIST
A list of the record data items involved.

4. FNEXT
The literal FIRST or NEXT, used with FIND3.

5. VAR
The name of database key variable, used with a FIND1.

6. SUPPRESS
A list of cosets to have currency updates suppressed.

Appendix C - CODE-A, a sample target language

The intermediate code generated by the ALF translator is designed to be either interpreted or translated into a higher level language. For pedagogic purposes only, this intermediate code, described in appendix B, is translated into an invented language called CODE-A. Real languages have features such as data item modes, data declarations, restrictions on the use of text strings, restricted control structures, and output formatting specifications. While the problems raised by these features would have to be faced in implementing a working relational system based on ALF, they are not the concern of this paper.

CODE-A is a language which exposes the structure of the generated programs in an easily understandable way, but which allows the inconvenient features of real languages to be ignored. It is essentially a reformatting of the intermediate code to make it more readable. Arithmetic and boolean expressions are in infix rather than prefix notation, and DML statements are as specified in (Codasyl 1971), rather than being specified as a set of assignment statements. User working area locations are specified in the form *record name_attribute name*. When they are required, the additional locations needed for queries where more than one variable ranges over the same location are specified in the form *relation variable_attribute name.*

Assignment statements are specified in the form

variable <- expression

There are two special variables, STATUS and CURRENT. STATUS is nonzero if the last DML command encountered either an error or some termination condition. CURRENT is the database key of the record found as a result of the last DML FIND command. Both these variables correspond to variables in the *system communication locations* defined in (Codasyl 1971).

There are two control structures in CODE-A, a WHILE loop and an IF statement.

The WHILE loop has the following form:

```
WHILE boolean expression DO
BEGINWHILE while loop id
:
:
EXITWHILE while loop id
:
:
ENDWHILE while loop id
```

The statements between the BEGINWHILE and the ENDWHILE are executed repeatedly until *boolean expression* becomes false.

There need not be an EXITWHILE in the loop, but if there is and it is executed, it causes control to be transferred to the statement following the ENDWHILE.

The IF statement has the following form :

```
IF boolean expression THEN
BEGINIF if statement id
:
:
ELSE
:
ENDIF if statement id
```

If *boolean expression* is true, the statements following the BEGINIF down to the optional ELSE or the ENDIF are executed. If it is false, then control passes to the statement following ELSE if ELSE is present, otherwise it passes to the statement following the ENDIF.

There are two statements, OUTPUT and PRINT, which correspond to ALF commands of the same name. They have the format:

```
OUTPUT expression[ , ... ] and
PRINT expression[ , ... ]
```

The expressions which appear in these statements have values corresponding to the targetlist values in the original ALF expression.

Appendix D - Schema Specification

The sub-language used for specifying data structure to the ALF translator mixes the functions of specifying the Internal and External schemata of (ANSI 1975). A full scale implementation of an ALF like language might well separate these functions.

In the following sub-language, first the relations, their attributes and their properties are defined, and then the cosets and their properties are defined. In defining the relations and attributes, virtual attributes are declared together with the coset used to materialise them.

The statements describing relations are:

relation-name (list of attribute names)

This is followed by a number of declarations chosen from the following

1. PRIMARY-KEY (primary key attributes)

2. CANDIDATE-KEYS ((attributes for candidate key-1) ...

3. CALC-KEY (attributes on which record is accessed using the CALC function)

4. INAREA area-name The area that the record is in.

5. SYSTEM-SET (system-coset-1 system-coset-2 ...
A list of the SYSTEM owned cosets the record is in.

6. VIRTUAL ((virtual-attribute-1 coset source-attribute-1)
(virtual-attribute-2 coset source-attribute-2)
A number of triples specifying each virtual data attribute in the relation and its source attribute.

These relation declarations are followed by the word "COSET-DECLARATIONS".

Each coset is declared in the following way

```
coset-name (owner member
            (list of owner attributes)
            (list of member attributes))
```

The owner and member record types are followed by the attribute in the owner and the corresponding (virtual) attributes in the member. Equality of values of these attributes in the relational model indicates coset membership in the corresponding network model.

A number of declarations may follow for each coset. The attributes appearing in the following declarations must be non-virtual attributes occurring in the member record of the coset being described.

1) SORTED (sort-attributes)

2) SORTED-INDEXED (attributes which are sorted and indexed)

3) SEARCH-KEYS ((attributes comprising search-key 1)
(attributes comprising search-key 2)

The coset declarations terminate with the word "END-COSET-DECLARATIONS".

Appendix E - Sample Computer Output

This Appendix shows the CODE-A generated for a number of sample ALF statements. The schema used is that defined in Section 4 of this report.

```
COMMENT
COMMENT   ************************************************************
COMMENT   ********************* APPENDIX E ***************************
COMMENT   ************************************************************
COMMENT   THE FOLLOWING EXAMPLES ILLUSTRATE THE CAPABILITIES OF ALF.
COMMENT   ************************************************************
COMMENT              TIMING INFORMATION.
COMMENT   THE FIRST TIME PRINTED IS THE TIME TO GENERATE THE
COMMENT   INTERNAL CODE FROM THE QUERY. THE SECOND IS THE FIRST
COMMENT   TIME, PLUS THE TIME TO OUTPUT THE CODE-A. GARBAGE COLLECTIONS
COMMENT   ARE INCLUDED IN THESE TIMES. GARBAGE COLLECTIONS COULD BE
COMMENT   REDUCED BY ALLOCATING MORE MEMORY.
COMMENT
COMMENT RETRIEVAL ON THE PRIMARY KEY OF THE COMP RELATION.
COMMENT  NO LOOPING CODE IS GENERATED.
 OUTPUT COMP.C#,COMP.CNAME,COMP.CLOC WHERE COMP.C#=123.
226 MILLISECONDS
START OF CODE-A
COMMENT START OF Q1 CODE
COMMENT TUPLE COUNT FOR Q1
N1 <-  0
COMP_C# <-  123
FIND COMP VIA CURRENT OF COMPSET USING COMP_C#   SUPPRESS
SS1,SS2,S1,COMPAREA CURRENCY UPDATES .
IF STATUS EQ 0  THEN
BEGINIF I1
COMP <-  CURRENT
GET COMP : COMP_C#,COMP_CNAME,COMP_CLOC
N1 <-  N1 + 1
OUTPUT COMP_C#,COMP_CNAME,COMP_CLOC
ENDIF I1
COMMENT END OF Q1 CODE
283 MILLISECONDS
```

```
INPUT ALF STATEMENT
COMMENT
COMMENT   A SIMPLE RETRIEVAL USING THE EMP RELATION ONLY.
COMMENT
     OUTPUT EMP.E# WHERE EMP.AGE GT 65 OR EMP.SAL < 8000 .
158 MILLISECONDS
START OF CODE-A
COMMENT START OF Q2 CODE
COMMENT TUPLE COUNT FOR Q2
N2 <-  0
FIND FIRST EMP RECORD OF EMPAREA AREA  SUPPRESS S2 CURRENCY
UPDATES .
COMMENT START OF EMP ( EMP ) LOOP
WHILE STATUS EQ 0  DO
BEGINWHILE W1
    EMP <-  CURRENT
    GET EMP : EMP_E#,EMP_SAL,EMP_AGE
    IF (EMP_SAL LT 8000 OR EMP_AGE GT 65))    THEN
    BEGINIF I2
    N2 <-  N2 + 1
    OUTPUT EMP_E#
    ENDIF I2
    FIND NEXT EMP RECORD OF EMPAREA AREA  SUPPRESS S2
    CURRENCY UPDATES .
ENDWHILE W1
COMMENT END OF Q2 CODE
233 MILLISECONDS
```

```
INPUT ALF STATEMENT
COMMENT
COMMENT   USE OF VIRTUAL ATTRIBUTES IN THE EMP RELATION.
COMMENT
     OUTPUT EMP.E# WHERE EMP.D# = "PERSONNEL" AND EMP.SAL < 8000 .
262 MILLISECONDS
START OF CODE-A
COMMENT START OF Q3 CODE
COMMENT TUPLE COUNT FOR Q3
N3 <-  0
FIND FIRST DEPT RECORD OF DEPAREA AREA  SUPPRESS S3,S1
CURRENCY UPDATES .
COMMENT START OF R1 ( DEPT ) LOOP
WHILE STATUS EQ 0  DO
BEGINWHILE W2
    R1 <-  CURRENT
    GET DEPT : DEPT_D#
    R1_D# <- DEPT_D#
    IF R1_D# EQ "PERSONNEL"  THEN
    BEGINIF I3
    FIND FIRST EMP RECORD OF S2 SET  SUPPRESS EMPAREA
    CURRENCY UPDATES .
    COMMENT START OF EMP ( EMP ) LOOP
    WHILE STATUS EQ 0  DO
    BEGINWHILE W3
        EMP <-  CURRENT
        GET EMP : EMP_E#,EMP_SAL
        IF EMP_SAL LT 8000  THEN
        BEGINIF I4
        N3 <-  N3 + 1
        OUTPUT EMP_E#
        ENDIF I4
        FIND NEXT EMP RECORD OF S2 SET  SUPPRESS EMPAREA
        CURRENCY UPDATES .
    ENDWHILE W3
    ENDIF I3
    FIND NEXT DEPT RECORD OF DEPAREA AREA  SUPPRESS S3,S1
    CURRENCY UPDATES .
ENDWHILE W2
COMMENT END OF Q3 CODE
488 MILLISECONDS
```

```
INPUT ALF STATEMENT
COMMENT
COMMENT   THIS QUERY ASKS FOR
COMMENT   EMPLOYEES WHOSE DEPARTMENTS ARE IN THE ACT.
COMMENT
     OUTPUT EMP.E#,DEPT.D# WHERE DEPT.C#=EMP.C# AND EMP.D# = DEPT.D#
 AND DEPT.DLOC="ACT".
454 MILLISECONDS
START OF CODE-A
COMMENT START OF Q4 CODE
COMMENT TUPLE COUNT FOR Q4
N4 <- 0
FIND FIRST COMP RECORD OF COMPSET SET  SUPPRESS SS1,SS2,
COMPAREA CURRENCY UPDATES .
COMMENT START OF R4 ( COMP ) LOOP
WHILE STATUS EQ 0  DO
BEGINWHILE W4
     R4 <- CURRENT
     DEPT_DLOC <- "ACT"
     FIND DEPT VIA CURRENT OF S1 USING DEPT_DLOC   SUPPRESS
     S3,DEPAREA CURRENCY UPDATES .
     COMMENT START OF DEPT ( DEPT ) LOOP
     WHILE STATUS EQ 0  DO
     BEGINWHILE W5
          DEPT <- CURRENT
          GET DEPT : DEPT_D#,DEPT_DLOC
          FIND FIRST EMP RECORD OF S2 SET  SUPPRESS EMPAREA
          CURRENCY UPDATES .
          COMMENT START OF EMP ( EMP ) LOOP
          WHILE STATUS EQ 0  DO
          BEGINWHILE W6
               EMP <- CURRENT
               GET EMP : EMP_E#
               N4 <- N4 + 1
               OUTPUT EMP_E#,DEPT_D#
               FIND NEXT EMP RECORD OF S2 SET  SUPPRESS
               EMPAREA CURRENCY UPDATES .
          ENDWHILE W6
          FIND NEXT DUPLICATE WITHIN S1 SET USING DEPT_DLOC
          SUPPRESS S3,DEPAREA CURRENCY UPDATES .
     ENDWHILE W5
     FIND NEXT COMP RECORD OF COMPSET SET  SUPPRESS SS1,SS2,
     COMPAREA CURRENCY UPDATES .
ENDWHILE W4
COMMENT END OF Q4 CODE
706 MILLISECONDS
```

```
INPUT ALF STATEMENT
COMMENT
COMMENT   THIS QUERY DOES NOT ASK FOR
COMMENT   EMPLOYEES WHOSE DEPARTMENTS ARE IN THE ACT.
COMMENT
   OUTPUT EMP.E#,DEPT.D# WHERE DEPT.C#=EMP.C# AND DEPT.DLOC="ACT".
348 MILLISECONDS
START OF CODE-A
COMMENT START OF Q5 CODE
COMMENT TUPLE COUNT FOR Q5
N5 <-  0
FIND FIRST COMP RECORD OF COMPSET SET  SUPPRESS SS1,SS2,
COMPAREA CURRENCY UPDATES .
COMMENT START OF R7 ( COMP ) LOOP
WHILE STATUS EQ 0  DO
BEGINWHILE W7
   R7 <-  CURRENT
   DEPT_DLOC <-  "ACT"
   FIND DEPT VIA CURRENT OF S1 USING DEPT_DLOC   SUPPRESS
   S2,S3,DEPAREA CURRENCY UPDATES .
   COMMENT START OF DEPT ( DEPT ) LOOP
   WHILE STATUS EQ 0  DO
   BEGINWHILE W8
      DEPT <-  CURRENT
      GET DEPT : DEPT_D#,DEPT_DLOC
      FIND FIRST DEPT RECORD OF S1 SET  SUPPRESS S3,
      DEPAREA CURRENCY UPDATES .
      COMMENT START OF R6 ( DEPT ) LOOP
      WHILE STATUS EQ 0  DO
      BEGINWHILE W9
         R6 <-  CURRENT
         FIND FIRST EMP RECORD OF S2 SET  SUPPRESS
         EMPAREA CURRENCY UPDATES .
         COMMENT START OF EMP ( EMP ) LOOP
         WHILE STATUS EQ 0  DO
         BEGINWHILE W10
            EMP <-  CURRENT
            GET EMP : EMP_E#
            N5 <-  N5 + 1
            OUTPUT EMP_E#,DEPT_D#
            FIND NEXT EMP RECORD OF S2 SET  SUPPRESS
            EMPAREA CURRENCY UPDATES .
         ENDWHILE W10
         FIND NEXT DEPT RECORD OF S1 SET  SUPPRESS S3,
         DEPAREA CURRENCY UPDATES .
      ENDWHILE W9
      FIND DEPT USING DEPT  SUPPRESS S2,S3,DEPAREA
      CURRENCY UPDATES .
      FIND NEXT DUPLICATE WITHIN S1 SET USING DEPT_DLOC
      SUPPRESS S2,S3,DEPAREA CURRENCY UPDATES .
   ENDWHILE W8
   FIND NEXT COMP RECORD OF COMPSET SET  SUPPRESS SS1,SS2,
   COMPAREA CURRENCY UPDATES .
ENDWHILE W7
COMMENT END OF Q5 CODE
648 MILLISECONDS
```

```
INPUT ALF STATEMENT
COMMENT
COMMENT   THIS QUERY DOES NOT ASK FOR
COMMENT   EMPLOYEES WHOSE DEPARTMENTS ARE IN THE ACT.
COMMENT
        RANGE OF D IS DEPT.
INPUT ALF STATEMENT
          OUTPUT EMP.E#,EMP.ENAME WHERE D.D#=EMP.D# AND D.DLOC="ACT".
375 MILLISECONDS
START OF CODE-A
COMMENT START OF Q6 CODE
COMMENT TUPLE COUNT FOR Q6
N6 <-  0
FIND FIRST DEPT RECORD OF DEPAREA AREA  SUPPRESS S2,S3,S1
CURRENCY UPDATES .
COMMENT START OF D ( DEPT ) LOOP
WHILE STATUS EQ 0  DO
BEGINWHILE W11
      D <-  CURRENT
      GET DEPT : DEPT_D#,DEPT_DLOC
      D_D# <- DEPT_D#
      D_DLOC <- DEPT_DLOC
      IF D_DLOC EQ "ACT"  THEN
      BEGINIF I5
      FIND FIRST DEPT RECORD OF DEPAREA AREA  SUPPRESS S3,S1
      CURRENCY UPDATES .
      COMMENT START OF R9 ( DEPT ) LOOP
      WHILE STATUS EQ 0  DO
      BEGINWHILE W12
            R9 <-  CURRENT
            GET DEPT : DEPT_D#
            R9_D# <-  DEPT_D#
            IF D_D# EQ R9_D#  THEN
            BEGINIF I6
            FIND FIRST EMP RECORD OF S2 SET  SUPPRESS EMPAREA
            CURRENCY UPDATES .
            COMMENT START OF EMP ( EMP ) LOOP
            WHILE STATUS EQ 0  DO
            BEGINWHILE W13
                  EMP <-  CURRENT
                  GET EMP : EMP_E#,EMP_ENAME
                  N6 <-  N6 + 1
                  OUTPUT EMP_E#,EMP_ENAME
                  FIND NEXT EMP RECORD OF S2 SET  SUPPRESS
                  EMPAREA CURRENCY UPDATES .
            ENDWHILE W13
            ENDIF I6
            FIND NEXT DEPT RECORD OF DEPAREA AREA  SUPPRESS S3,
            S1 CURRENCY UPDATES .
      ENDWHILE W12
      ENDIF I5
      FIND DEPT USING D  SUPPRESS S2,S3,S1 CURRENCY UPDATES .
      FIND NEXT DEPT RECORD OF DEPAREA AREA  SUPPRESS S2,S3,
      S1 CURRENCY UPDATES .
ENDWHILE W11
COMMENT END OF Q6 CODE
661 MILLISECONDS
```

```
INPUT ALF STATEMENT
COMMENT
COMMENT   DEPARTMENTS WHOSE BUDGET IS GREATER THAN THE AVERAGE
COMMENT   DEPARTMENT BUDGET. (IE FOR ALL COMPANIES)
COMMENT
     RANGE OF D1,D2 IS DEPT.
INPUT ALF STATEMENT
     OUTPUT D1.D# WHERE
     D1.BUDGET GT AVG(D2.BUDGET).
210 MILLISECONDS
START OF CODE-A
COMMENT START OF Q7 CODE
COMMENT TUPLE COUNT FOR Q7
N7 <-  0
COMMENT START OF Q8 CODE
COMMENT TUPLE COUNT FOR Q8
N8 <-  0
Q8 <-  0
FIND FIRST DEPT RECORD OF DEPAREA AREA  SUPPRESS S2,S3,S1
CURRENCY UPDATES .
COMMENT START OF D2 ( DEPT ) LOOP
WHILE STATUS EQ 0   DO
BEGINWHILE W14
    D2 <-  CURRENT
    GET DEPT : DEPT_BUDGET
    D2_BUDGET <-  DEPT_BUDGET
    N8 <-  N8 + 1
    Q8 <-  Q8 + D2_BUDGET
    FIND NEXT DEPT RECORD OF DEPAREA AREA  SUPPRESS S2,S3,
    S1 CURRENCY UPDATES .
ENDWHILE W14
IF N8 NE 0 THEN
BEGINIF I7
Q8 <-  Q8 / N8
ENDIF I7
COMMENT END OF Q8 CODE
FIND FIRST DEPT RECORD OF DEPAREA AREA  SUPPRESS S2,S3,S1
CURRENCY UPDATES .
COMMENT START OF D1 ( DEPT ) LOOP
WHILE STATUS EQ 0   DO
BEGINWHILE W15
    D1 <-  CURRENT
    GET DEPT : DEPT_D#,DEPT_BUDGET
    D1_D# <-  DEPT_D#
    D1_BUDGET <-  DEPT_BUDGET
    IF D1_BUDGET GT Q8  THEN
    BEGINIF I8
    N7 <-  N7 + 1
    OUTPUT D1_D#
    ENDIF I8
    FIND NEXT DEPT RECORD OF DEPAREA AREA  SUPPRESS S2,S3,
    S1 CURRENCY UPDATES .
ENDWHILE W15
COMMENT END OF Q7 CODE
472 MILLISECONDS
```

```
INPUT ALF STATEMENT
COMMENT
COMMENT   DEPARTMENTS WHOSE BUDGET IS GREATER THAN THE AVERAGE
COMMENT   DEPARTMENTAL BUDGET FOR THEIR COMPANY.
COMMENT
      OUTPUT D1.D# WHERE
      D1.BUDGET GT AVG(D2.BUDGET WHERE D1.C#=D2.C#).
331 MILLISECONDS
START OF CODE-A
COMMENT START OF Q9 CODE
COMMENT TUPLE COUNT FOR Q9
N9 <-  0
FIND FIRST COMP RECORD OF COMPSET SET  SUPPRESS SS1,SS2,
COMPAREA CURRENCY UPDATES .
COMMENT START OF R10 ( COMP ) LOOP
WHILE STATUS EQ 0  DO
BEGINWHILE W16
    R10 <-  CURRENT
    COMMENT START OF Q10 CODE
    COMMENT TUPLE COUNT FOR Q10
    N10 <-  0
    Q10 <-  0
    FIND FIRST DEPT RECORD OF S1 SET  SUPPRESS S2,S3,
    DEPAREA CURRENCY UPDATES .
    COMMENT START OF D2 ( DEPT ) LOOP
    WHILE STATUS EQ 0  DO
    BEGINWHILE W17
        D2 <-  CURRENT
        GET DEPT : DEPT_BUDGET
        D2_BUDGET <-  DEPT_BUDGET
        N10 <-  N10 + 1
        Q10 <-  Q10 + D2_BUDGET
        FIND NEXT DEPT RECORD OF S1 SET  SUPPRESS S2,S3,
        DEPAREA CURRENCY UPDATES .
    ENDWHILE W17
    IF N10 NE 0  THEN
    BEGINIF I9
    Q10 <-  Q10 / N10
    ENDIF I9
    COMMENT END OF Q10 CODE
    FIND FIRST DEPT RECORD OF S1 SET  SUPPRESS S2,S3,
    DEPAREA CURRENCY UPDATES .
    COMMENT START OF D1 ( DEPT ) LOOP
    WHILE STATUS EQ 0  DO
    BEGINWHILE W18
        D1 <-  CURRENT
        GET DEPT : DEPT_D#,DEPT_BUDGET
        D1_D# <-  DEPT_D#
        D1_BUDGET <-  DEPT_BUDGET
        IF D1_BUDGET GT Q10  THEN
        BEGINIF I10
        N9 <-  N9 + 1
        OUTPUT D1_D#
        ENDIF I10
        FIND NEXT DEPT RECORD OF S1 SET  SUPPRESS S2,S3,
        DEPAREA CURRENCY UPDATES .
    ENDWHILE W18
    FIND NEXT COMP RECORD OF COMPSET SET  SUPPRESS SS1,SS2,
    COMPAREA CURRENCY UPDATES .
ENDWHILE W16
COMMENT END OF Q9 CODE
```

```
632 MILLISECONDS
```

```
INPUT ALF STATEMENT
COMMENT
COMMENT THE FOLLOWING FOUR QUERIES SHOW BOOLEAN TEST OPTIMISATION.
COMMENT
     OUTPUT COMP.C#,COMP.CLOC WHERE COMP.CLOC="ACT"
 AND COMP.C# NE 123 AND
     AVG(DEPT.BUDGET WHERE DEPT.C#=COMP.C#) GT 50000 .
371 MILLISECONDS
START OF CODE-A
COMMENT START OF Q11 CODE
COMMENT TUPLE COUNT FOR Q11
N11 <-  0
COMP_CLOC <- "ACT"
FIND COMP VIA CURRENT OF COMPSET USING COMP_CLOC   SUPPRESS
SS1,SS2,S1,COMPAREA CURRENCY UPDATES .
COMMENT START OF COMP ( COMP ) LOOP
WHILE STATUS EQ 0  DO
BEGINWHILE W19
    COMP <-  CURRENT
    GET COMP : COMP_C#,COMP_CLOC
    IF COMP_C# NE 123  THEN
    BEGINIF I11
    COMMENT START OF Q12 CODE
    COMMENT TUPLE COUNT FOR Q12
    N12 <-  0
    Q12 <-  0
    FIND FIRST DEPT RECORD OF S1 SET  SUPPRESS S2,S3,
    DEPAREA CURRENCY UPDATES .
    COMMENT START OF DEPT ( DEPT ) LOOP
    WHILE STATUS EQ 0  DO
    BEGINWHILE W20
        DEPT <-  CURRENT
        GET DEPT : DEPT_BUDGET
        N12 <-  N12 + 1
        Q12 <-  Q12 + DEPT_BUDGET
        FIND NEXT DEPT RECORD OF S1 SET  SUPPRESS S2,S3,
        DEPAREA CURRENCY UPDATES .
    ENDWHILE W20
    IF N12 NE 0  THEN
    BEGINIF I12
    Q12 <-  Q12 / N12
    ENDIF I12
    COMMENT END OF Q12 CODE
    IF Q12 GT 50000  THEN
    BEGINIF I13
    N11 <-  N11 + 1
    OUTPUT COMP_C#,COMP_CLOC
    ENDIF I13
    ENDIF I11
    FIND NEXT DUPLICATE WITHIN COMPSET SET USING COMP_CLOC
    SUPPRESS SS1,SS2,S1,COMPAREA CURRENCY UPDATES .
ENDWHILE W19
COMMENT END OF Q11 CODE
641 MILLISECONDS
```

```
INPUT ALF STATEMENT
    OUTPUT COMP.C#,COMP.CLOC WHERE COMP.CLOC="ACT"
 AND COMP.C# NE 123 OR
    AVG(DEPT.BUDGET WHERE DEPT.C#=COMP.C#) GT 50000 .
439 MILLISECONDS
START OF CODE-A
COMMENT START OF Q13 CODE
COMMENT TUPLE COUNT FOR Q13
N13 <-  0
FIND FIRST COMP RECORD OF COMPSET SET  SUPPRESS SS1,SS2,S1,
COMPAREA CURRENCY UPDATES .
COMMENT START OF COMP ( COMP ) LOOP
WHILE STATUS EQ 0  DO
BEGINWHILE W21
    COMP <-  CURRENT
    GET COMP : COMP_C#,COMP_CLOC
    B1 <-  TRUE
    IF (COMP_C# EQ 123 OR COMP_CLOC NE "ACT"))   THEN
    BEGINIF I14
    COMMENT START OF Q14 CODE
    COMMENT TUPLE COUNT FOR Q14
    N14 <-  0
    Q14 <-  0
    FIND FIRST DEPT RECORD OF S1 SET  SUPPRESS S2,S3,
    DEPAREA CURRENCY UPDATES .
    COMMENT START OF DEPT ( DEPT ) LOOP
    WHILE STATUS EQ 0  DO
    BEGINWHILE W22
        DEPT <-  CURRENT
        GET DEPT : DEPT_BUDGET
        N14 <-  N14 + 1
        Q14 <-  Q14 + DEPT_BUDGET
        FIND NEXT DEPT RECORD OF S1 SET  SUPPRESS S2,S3,
        DEPAREA CURRENCY UPDATES .
    ENDWHILE W22
    IF N14 NE 0  THEN
    BEGINIF I15
    Q14 <-  Q14 / N14
    ENDIF I15
    COMMENT END OF Q14 CODE
    IF Q14 LE 50000  THEN
    BEGINIF I16
    B1 <-  FALSE
    ENDIF I16
    ENDIF I14
    IF B1 EQ TRUE  THEN
    BEGINIF I17
    N13 <-  N13 + 1
    OUTPUT COMP_C#,COMP_CLOC
    ENDIF I17
    FIND NEXT COMP RECORD OF COMPSET SET  SUPPRESS SS1,SS2,
    S1,COMPAREA CURRENCY UPDATES .
ENDWHILE W21
COMMENT END OF Q13 CODE
719 MILLISECONDS
```

```
INPUT ALF STATEMENT
      OUTPUT COMP.C#,COMP.CLOC WHERE COMP.CLOC NE "ACT"
 AND COMP.C# NE 123 AND
      AVG(DEPT.BUDGET WHERE DEPT.C#=COMP.C#) GT 50000 .
374 MILLISECONDS
START OF CODE-A
COMMENT START OF Q15 CODE
COMMENT TUPLE COUNT FOR Q15
N15 <-  0
FIND FIRST COMP RECORD OF COMPSET SET  SUPPRESS SS1,SS2,S1,
COMPAREA CURRENCY UPDATES .
COMMENT START OF COMP ( COMP ) LOOP
WHILE STATUS EQ 0  DO
BEGINWHILE W23
    COMP <-  CURRENT
    GET COMP : COMP_C#,COMP_CLOC
    IF (COMP_CLOC NE "ACT" AND COMP_C# NE 123))   THEN
    BEGINIF I18
    COMMENT START OF Q16 CODE
    COMMENT TUPLE COUNT FOR Q16
    N16 <-  0
    Q16 <-  0
    FIND FIRST DEPT RECORD OF S1 SET  SUPPRESS S2,S3,
    DEPAREA CURRENCY UPDATES .
    COMMENT START OF DEPT ( DEPT ) LOOP
    WHILE STATUS EQ 0  DO
    BEGINWHILE W24
        DEPT <-  CURRENT
        GET DEPT : DEPT_BUDGET
        N16 <-  N16 + 1
        Q16 <-  Q16 + DEPT_BUDGET
        FIND NEXT DEPT RECORD OF S1 SET  SUPPRESS S2,S3,
        DEPAREA CURRENCY UPDATES .
    ENDWHILE W24
    IF N16 NE 0  THEN
    BEGINIF I19
    Q16 <-  Q16 / N16
    ENDIF I19
    COMMENT END OF Q16 CODE
    IF Q16 GT 50000  THEN
    BEGINIF I20
    N15 <-  N15 + 1
    OUTPUT COMP_C#,COMP_CLOC
    ENDIF I20
    ENDIF I18
    FIND NEXT COMP RECORD OF COMPSET SET  SUPPRESS SS1,SS2,
    S1,COMPAREA CURRENCY UPDATES .
ENDWHILE W23
COMMENT END OF Q15 CODE
641 MILLISECONDS
```

```
INPUT ALF STATEMENT
     OUTPUT COMP.C#,COMP.CLOC WHERE COMP.CLOC NE "ACT"
 AND COMP.C# NE 123 OR
     AVG(DEPT.BUDGET WHERE DEPT.C#=COMP.C#) GT 50000 .
414 MILLISECONDS
START OF CODE-A
COMMENT START OF Q17 CODE
COMMENT TUPLE COUNT FOR Q17
N17 <-  0
FIND FIRST COMP RECORD OF COMPSET SET  SUPPRESS SS1,SS2,S1,
COMPAREA CURRENCY UPDATES .
COMMENT START OF COMP ( COMP ) LOOP
WHILE STATUS EQ 0  DO
BEGINWHILE W25
    COMP <-  CURRENT
    GET COMP : COMP_C#,COMP_CLOC
    B2 <-  TRUE
    IF (COMP_C# EQ 123 OR COMP_CLOC EQ "ACT"))   THEN
    BEGINIF I21
    COMMENT START OF Q18 CODE
    COMMENT TUPLE COUNT FOR Q18
    N18 <-  0
    Q18 <-  0
    FIND FIRST DEPT RECORD OF S1 SET  SUPPRESS S2,S3,
    DEPAREA CURRENCY UPDATES .
    COMMENT START OF DEPT ( DEPT ) LOOP
    WHILE STATUS EQ 0  DO
    BEGINWHILE W26
        DEPT <-  CURRENT
        GET DEPT : DEPT_BUDGET
        N18 <-  N18 + 1
        Q18 <-  Q18 + DEPT_BUDGET
        FIND NEXT DEPT RECORD OF S1 SET  SUPPRESS S2,S3,
        DEPAREA CURRENCY UPDATES .
    ENDWHILE W26
    IF N18 NE 0  THEN
    BEGINIF I22
    Q18 <-  Q18 / N18
    ENDIF I22
    COMMENT END OF Q18 CODE
    IF Q18 LE 50000  THEN
    BEGINIF I23
    B2 <-  FALSE
    ENDIF I23
    ENDIF I21
    IF B2 EQ TRUE  THEN
    BEGINIF I24
    N17 <-  N17 + 1
    OUTPUT COMP_C#,COMP_CLOC
    ENDIF I24
    FIND NEXT COMP RECORD OF COMPSET SET  SUPPRESS SS1,SS2,
    S1,COMPAREA CURRENCY UPDATES .
ENDWHILE W25
COMMENT END OF Q17 CODE
695 MILLISECONDS
```

```
INPUT ALF STATEMENT
COMMENT
COMMENT TWO AGGREGATE FUNCTIONS, WHICH WILL BE COMPUTED TOGETHER.
COMMENT
COMMENT THIS QUERY ASKS FOR DEPARTMENTS IN COMPANIES
COMMENT WHERE THE MAXIMUM DEPARTMENTAL BUDGET OF ACT
COMMENT DEPARTMENTS IS GT 4000000  AND WHERE THE AVERAGE
COMMENT DEPARTMENTAL BUDGET OF ACT DEPARTMENTS IS LT 2000000
COMMENT
      OUTPUT DEPT.D# WHERE
      DEPT.C# EQ COMP.C# AND
      AVG(D1.BUDGET WHERE D1.C# EQ COMP.C#
       AND D1.DLOC EQ "ACT") GT 2000000
      AND
      MAX(D2.BUDGET WHERE D2.C# EQ COMP.C#
      AND D2.DLOC EQ "ACT") LT 4000000
      .
793 MILLISECONDS
START OF CODE-A
COMMENT START OF Q19 CODE
COMMENT TUPLE COUNT FOR Q19
N19 <-  0
FIND FIRST COMP RECORD OF COMPSET SET  SUPPRESS SS1,SS2,
COMPAREA CURRENCY UPDATES .
COMMENT START OF COMP ( COMP ) LOOP
WHILE STATUS EQ 0  DO
BEGINWHILE W27
    COMP <-  CURRENT
    COMMENT START OF Q21 CODE
    COMMENT TUPLE COUNT FOR Q21
    N20 <-  0
    Q21 <-  LOW-VALUES
    Q20 <-  0
    DEPT_DLOC <-  "ACT"
    FIND DEPT VIA CURRENT OF S1 USING DEPT_DLOC   SUPPRESS
    S2,S3,DEPAREA CURRENCY UPDATES .
    COMMENT START OF D2 ( DEPT ) LOOP
    WHILE STATUS EQ 0  DO
    BEGINWHILE W28
        D2 <-  CURRENT
        GET DEPT : DEPT_BUDGET,DEPT_DLOC
        D2_BUDGET <-  DEPT_BUDGET
        D2_DLOC <-  DEPT_DLOC
        N20 <-  N20 + 1
        Q21 <-  MAXIMUM(Q21,D2_BUDGET)
        Q20 <-  Q20 + D2_BUDGET
        FIND NEXT DUPLICATE WITHIN S1 SET USING DEPT_DLOC
        SUPPRESS S2,S3,DEPAREA CURRENCY UPDATES .
    ENDWHILE W28
    IF N20 NE 0  THEN
    BEGINIF I25
    Q20 <-  Q20 / N20
    ENDIF I25
    COMMENT END OF Q21 CODE
    IF Q21 LT 4000000  THEN
    BEGINIF I26
    IF Q20 GT 2000000  THEN
    BEGINIF I27
    FIND FIRST DEPT RECORD OF S1 SET  SUPPRESS S2,S3,
    DEPAREA CURRENCY UPDATES .
    COMMENT START OF DEPT ( DEPT ) LOOP
    WHILE STATUS EQ 0  DO
    BEGINWHILE W29
        DEPT <-  CURRENT
        GET DEPT : DEPT_D#
        N19 <-  N19 + 1
        OUTPUT DEPT_D#
        FIND NEXT DEPT RECORD OF S1 SET  SUPPRESS S2,S3,
        DEPAREA CURRENCY UPDATES .
    ENDWHILE W29
    ENDIF I27
    ENDIF I26
    FIND NEXT COMP RECORD OF COMPSET SET  SUPPRESS SS1,SS2,
    COMPAREA CURRENCY UPDATES .
ENDWHILE W27
COMMENT END OF Q19 CODE
1008 MILLISECONDS
```

```
INPUT ALF STATEMENT
COMMENT
COMMENT  SUPPLIERS WHO SUPPLY NO PARTS
COMMENT
    RANGE OF S,P,SP IS SUPPLIER,PART,SUPPLY .
INPUT ALF STATEMENT
    OUTPUT S.S# WHERE
      ALL(SP.S# WHERE  SP.S# NE S.S#)=TRUE.
354 MILLISECONDS
START OF CODE-A
COMMENT START OF Q22 CODE
COMMENT TUPLE COUNT FOR Q22
N21 <- 0
FIND FIRST SUPPLIER RECORD OF SUPPLIER-SET SET  SUPPRESS
SUPPLIES,SUPPAREA CURRENCY UPDATES .
COMMENT START OF S ( SUPPLIER ) LOOP
WHILE STATUS EQ 0  DO
BEGINWHILE W30
S <-  CURRENT
    GET SUPPLIER : SUPPLIER_S#
    S_S# <- SUPPLIER_S#
    COMMENT START OF Q23 CCDE
    COMMENT TUPLE COUNT FOR Q23
    N22 <- 0
    Q23 <-  FALSE
    FIND FIRST SUPPLY RECORD OF SUPPLIES SET  SUPPRESS
    IS-SUPPLIED-BY,PARTAREA CURRENCY UPDATES .
    COMMENT START OF SP ( SUPPLY ) LOOP
    WHILE STATUS EQ 0  DO
    BEGINWHILE W31
        SP <-  CURRENT
        N22 <-  N22 + 1
        Q23 <-  TRUE
        EXITWHILE W31
        FIND NEXT SUPPLY RECORD OF SUPPLIES SET  SUPPRESS
        IS-SUPPLIED-BY,PARTAREA CURRENCY UPDATES .
    ENDWHILE W31
    COMMENT END OF Q23 CODE
    IF Q23 NE TRUE  THEN
    BEGINIF I28
    N21 <-  N21 + 1
    OUTPUT S_S#
    ENDIF I28
    FIND NEXT SUPPLIER RECORD OF SUPPLIER-SET SET  SUPPRESS
    SUPPLIES,SUPPAREA CURRENCY UPDATES .
ENDWHILE W30
COMMENT END OF Q22 CODE
486 MILLISECONDS
```

```
INPUT ALF STATEMENT
COMMENT
COMMENT   SUPPLIERS WHO SUPPLY ALL PARTS
COMMENT
 OUTPUT S.S# WHERE
     ALL(P.P# WHERE
         EXISTS(SP.S# WHERE
         SP.S#=S.S# AND SP.P# =P.P#
         ) IS TRUE
    ) IS TRUE.
560 MILLISECONDS
START OF CODE-A
COMMENT START OF Q24 CODE
COMMENT TUPLE COUNT FOR Q24
N23 <-  0
FIND FIRST SUPPLIER RECORD OF SUPPLIER-SET SET  SUPPRESS
SUPPLIES,SUPPAREA CURRENCY UPDATES .
COMMENT START OF S ( SUPPLIER ) LOOP
WHILE STATUS EQ 0  DO
BEGINWHILE W32
    S <-  CURRENT
    GET SUPPLIER : SUPPLIER_S#
    S_S# <-  SUPPLIER_S#
    COMMENT START OF Q25 CODE
    COMMENT TUPLE COUNT FOR Q25
    N24 <-  0
    Q25 <-  FALSE
    FIND FIRST PART RECORD OF PARTSET SET  SUPPRESS
    IS-SUPPLIED-BY,PARTAREA CURRENCY UPDATES .
    COMMENT START OF P ( PART ) LOOP
    WHILE STATUS EQ 0  DO
    BEGINWHILE W33
        P <-  CURRENT
        GET PART : PART_P#
        P_P# <-  PART_P#
        COMMENT START OF Q26 CODE
        COMMENT TUPLE COUNT FOR Q26
        N25 <-  0
        Q26 <-  FALSE
        FIND FIRST SUPPLY RECORD OF IS-SUPPLIED-BY SET
        SUPPRESS PARTAREA CURRENCY UPDATES .
        COMMENT START OF SP ( SUPPLY ) LOOP
        WHILE STATUS EQ 0  DO
        BEGINWHILE W34
            SP <-  CURRENT
            FIND OWNER RECORD OF SUPPLIES SET  SUPPRESS
            SUPPLIER-SET,SUPPAREA CURRENCY UPDATES .
            IF CURRENT EQ S  THEN
            BEGINIF I29
            N25 <-  N25 + 1
            Q26 <-  TRUE
            EXITWHILE W34
            ENDIF I29
            FIND NEXT SUPPLY RECORD OF IS-SUPPLIED-BY SET
            SUPPRESS PARTAREA CURRENCY UPDATES .
        ENDWHILE W34
        COMMENT END OF Q26 CODE
        IF Q26 NE TRUE  THEN
        BEGINIF I30
        N24 <-  N24 + 1
        Q25 <-  TRUE
```

```
            EXITWHILE W33
            ENDIF I30
            FIND NEXT PART RECORD OF PARTSET SET  SUPPRESS
            IS-SUPPLIED-BY,PARTAREA CURRENCY UPDATES .
        ENDWHILE W33
        COMMENT END OF Q25 CODE
        IF Q25 NE TRUE  THEN
        BEGINIF I31
        N23 <-  N23 + 1
        OUTPUT S_S#
        ENDIF I31
        FIND NEXT SUPPLIER RECORD OF SUPPLIER-SET SET  SUPPRESS
        SUPPLIES,SUPPAREA CURRENCY UPDATES .
    ENDWHILE W32
    COMMENT END OF Q24 CODE
    888 MILLISECONDS
```

```
INPUT ALF STATEMENT
COMMENT
COMMENT SUPPLIERS WHO SUPPLY A GREEN PART (IE AT LEAST ONE)
COMMENT
 OUTPUT S.S# WHERE
       EXISTS(P.P# WHERE
           P.COLOUR="GREEN" AND
           EXISTS(SP.P# WHERE SP.S#=S.S# AND SP.P#=P.P#) IS TRUE
       ) IS TRUE.
510 MILLISECONDS
START OF CODE-A
COMMENT START OF Q27 CODE
COMMENT TUPLE COUNT FOR Q27
N26 <-  0
FIND FIRST SUPPLIER RECORD OF SUPPLIER-SET SET  SUPPRESS
SUPPLIES,SUPPAREA CURRENCY UPDATES .
COMMENT START OF S ( SUPPLIER ) LOOP
WHILE STATUS EQ 0  DO
BEGINWHILE W35
    S <-  CURRENT
    GET SUPPLIER : SUPPLIER_S#
    S_S# <-  SUPPLIER_S#
    COMMENT START OF Q28 CODE
    COMMENT TUPLE COUNT FOR Q28
    N27 <-  0
    Q28 <-  FALSE
    PART_COLOUR <-  "GREEN"
    FIND PART VIA CURRENT OF PARTSET USING PART_COLOUR
    SUPPRESS IS-SUPPLIED-BY,PARTAREA CURRENCY UPDATES .
    COMMENT START OF P ( PART ) LOOP
    WHILE STATUS EQ 0  DO
    BEGINWHILE W36
        P <-  CURRENT
        GET PART : PART_P#,PART_COLOUR
        P_P# <-  PART_P#
        P_COLOUR <-  PART_COLOUR
        COMMENT START OF Q29 CODE
        COMMENT TUPLE COUNT FOR Q29
        N28 <-  0
        Q29 <-  FALSE
        FIND FIRST SUPPLY RECORD OF IS-SUPPLIED-BY SET
        SUPPRESS PARTAREA CURRENCY UPDATES .
        COMMENT START OF SP ( SUPPLY ) LOOP
        WHILE STATUS EQ 0  DO
        BEGINWHILE W37
            SP <-  CURRENT
            FIND OWNER RECORD OF SUPPLIES SET  SUPPRESS
            SUPPLIER-SET,SUPPAREA CURRENCY UPDATES .
            IF CURRENT EQ S  THEN
            BEGINIF I32
            N28 <-  N28 + 1
            Q29 <-  TRUE
            EXITWHILE W37
            ENDIF I32
            FIND NEXT SUPPLY RECORD OF IS-SUPPLIED-BY SET
            SUPPRESS PARTAREA CURRENCY UPDATES .
        ENDWHILE W37
        COMMENT END OF Q29 CODE
        IF Q29 EQ TRUE  THEN
        BEGINIF I33
        N27 <-  N27 + 1
```

```
            Q28 <-  TRUE
            EXITWHILE W36
            ENDIF I33
            FIND NEXT DUPLICATE WITHIN PARTSET SET USING
            PART_COLOUR   SUPPRESS IS-SUPPLIED-BY,PARTAREA
            CURRENCY UPDATES .
        ENDWHILE W36
        COMMENT END OF Q28 CODE
        IF Q28 EQ TRUE  THEN
        BEGINIF I34
        N26 <-  N26 + 1
        OUTPUT S_S#
        ENDIF I34
        FIND NEXT SUPPLIER RECORD OF SUPPLIER-SET SET  SUPPRESS
        SUPPLIES,SUPPAREA CURRENCY UPDATES .
    ENDWHILE W35
    COMMENT END OF Q27 CODE
    851 MILLISECONDS
```

```
INPUT ALF STATEMENT
COMMENT
COMMENT ANOTHER VERSION, WITH EXISTENTIAL QUANTIFIERS COMBINED.
COMMENT
 OUTPUT S.S# WHERE
      EXISTS(P.P#,SP.P# WHERE
          P.COLOUR="GREEN" AND
          SP.S#=S.S# AND SP.P#=P.P#
      ) IS TRUE.
409 MILLISECONDS
START OF CODE-A
COMMENT START OF Q30 CODE
COMMENT TUPLE COUNT FOR Q30
N29 <-  0
FIND FIRST SUPPLIER RECORD OF SUPPLIER-SET SET  SUPPRESS
SUPPLIES,SUPPAREA CURRENCY UPDATES .
COMMENT START OF S ( SUPPLIER ) LOOP
WHILE STATUS EQ 0  DO
BEGINWHILE W38
    S <-  CURRENT
    GET SUPPLIER : SUPPLIER_S#
    S_S# <-  SUPPLIER_S#
    COMMENT START OF Q31 CODE
    COMMENT TUPLE COUNT FOR Q31
    N30 <-  0
    Q31 <-  FALSE
    FIND FIRST SUPPLY RECORD OF SUPPLIES SET  SUPPRESS
    PARTAREA CURRENCY UPDATES .
    COMMENT START OF SP ( SUPPLY ) LOOP
    WHILE STATUS EQ 0  DO
    BEGINWHILE W39
        SP <-  CURRENT
        FIND OWNER RECORD OF IS-SUPPLIED-BY SET  SUPPRESS
        PARTSET,PARTAREA CURRENCY UPDATES .
        P <-  CURRENT
        GET PART : PART_P#,PART_COLOUR
        P_P# <-  PART_P#
        P_COLOUR <-  PART_COLOUR
        IF P_COLOUR EQ "GREEN"  THEN
        BEGINIF I35
        N30 <-  N30 + 1
        Q31 <-  TRUE
        EXITWHILE W39
        ENDIF I35
        FIND NEXT SUPPLY RECORD OF SUPPLIES SET  SUPPRESS
        PARTAREA CURRENCY UPDATES .
    ENDWHILE W39
    COMMENT END OF Q31 CODE
    IF Q31 EQ TRUE  THEN
    BEGINIF I36
    N29 <-  N29 + 1
    OUTPUT S_S#
    ENDIF I36
    FIND NEXT SUPPLIER RECORD OF SUPPLIER-SET SET  SUPPRESS
    SUPPLIES,SUPPAREA CURRENCY UPDATES .
ENDWHILE W38
COMMENT END OF Q30 CODE
679 MILLISECONDS
```

```
INPUT ALF STATEMENT
COMMENT
COMMENT SUPPLIERS WHO SUPPLY ONLY GREEN PARTS
COMMENT
    OUTPUT S.S# WHERE
    ALL(P.P# :
      EXISTS(SP.P# :
      SP.S#=S.S# AND SP.P#=P.P#) IS TRUE IMPLIES P.COLOUR = "GREEN"
    )=TRUE.
616 MILLISECONDS
START OF CODE-A
COMMENT START OF Q32 CODE
COMMENT TUPLE COUNT FOR Q32
N31 <-  0
FIND FIRST SUPPLIER RECORD OF SUPPLIER-SET SET  SUPPRESS
SUPPLIES,SUPPAREA CURRENCY UPDATES .
COMMENT START OF S ( SUPPLIER ) LOOP
WHILE STATUS EQ 0  DO
BEGINWHILE W40
    S <-  CURRENT
    GET SUPPLIER : SUPPLIER_S#
    S_S# <-  SUPPLIER_S#
    COMMENT START OF Q33 CODE
    COMMENT TUPLE COUNT FOR Q33
    N32 <-  0
    Q33 <-  FALSE
    FIND FIRST PART RECORD OF PARTSET SET  SUPPRESS
    IS-SUPPLIED-BY,PARTAREA CURRENCY UPDATES .
    COMMENT START OF P ( PART ) LOOP
    WHILE STATUS EQ 0  DO
    BEGINWHILE W41
        P <-  CURRENT
        GET PART : PART_P#,PART_COLOUR
        P_P# <-  PART_P#
        P_COLOUR <-  PART_COLOUR
        IF P_COLOUR NE "GREEN"  THEN
        BEGINIF I37
        COMMENT START OF Q34 CODE
        COMMENT TUPLE COUNT FOR Q34
        N33 <-  0
        Q34 <-  FALSE
        FIND FIRST SUPPLY RECORD OF IS-SUPPLIED-BY SET
        SUPPRESS PARTAREA CURRENCY UPDATES .
        COMMENT START OF SP ( SUPPLY ) LOOP
        WHILE STATUS EQ 0  DO
        BEGINWHILE W42
            SP <-  CURRENT
            FIND OWNER RECORD OF SUPPLIES SET  SUPPRESS
            SUPPLIER-SET,SUPPAREA CURRENCY UPDATES .
            IF CURRENT EQ S  THEN
            BEGINIF I38
            N33 <-  N33 + 1
            Q34 <-  TRUE
            EXITWHILE W42
            ENDIF I38
            FIND NEXT SUPPLY RECORD OF IS-SUPPLIED-BY SET
            SUPPRESS PARTAREA CURRENCY UPDATES .
        ENDWHILE W42
        COMMENT END OF Q34 CODE
        IF Q34 EQ TRUE  THEN
        BEGINIF I39
```

```
            N32 <-  N32 + 1
            Q33 <-  TRUE
            EXITWHILE W41
            ENDIF I39
            ENDIF I37
            FIND NEXT PART RECORD OF PARTSET SET  SUPPRESS
         IS-SUPPLIED-BY,PARTAREA CURRENCY UPDATES .
        ENDWHILE W41
        COMMENT END OF Q33 CODE
        IF Q33 NE TRUE  THEN
        BEGINIF I40
        N31 <-  N31 + 1
        OUTPUT S_S#
        ENDIF I40
        FIND NEXT SUPPLIER RECORD OF SUPPLIER-SET SET  SUPPRESS
        SUPPLIES,SUPPAREA CURRENCY UPDATES .
    ENDWHILE W40
    COMMENT END OF Q32 CODE
    958 MILLISECONDS
```

```
INPUT ALF STATEMENT
COMMENT
COMMENT SUPPLIERS WHO SUPPLY ALL THE GREEN PARTS
COMMENT
     OUTPUT S.S# :
     ALL(P.P# :
         P.COLOUR="GREEN" IMPLIES
         EXISTS(SP.P# : SP.S#=S.S# AND SP.P# = P.P#) IS TRUE
     )
     IS TRUE.
497 MILLISECONDS
START OF CODE-A
COMMENT START OF Q35 CODE
COMMENT TUPLE COUNT FOR Q35
N34 <-  0
FIND FIRST SUPPLIER RECORD OF SUPPLIER-SET SET  SUPPRESS
SUPPLIES,SUPPAREA CURRENCY UPDATES .
COMMENT START OF S ( SUPPLIER ) LOOP
WHILE STATUS EQ 0  DO
BEGINWHILE W43
    S <-  CURRENT
    GET SUPPLIER : SUPPLIER_S#
    S_S# <-  SUPPLIER_S#
    COMMENT START OF Q36 CODE
    COMMENT TUPLE COUNT FOR Q36
    N35 <-  0
    Q36 <-  FALSE
    PART_COLOUR <-  "GREEN"
    FIND PART VIA CURRENT OF PARTSET USING PART_COLOUR
    SUPPRESS IS-SUPPLIED-BY,PARTAREA CURRENCY UPDATES .
    COMMENT START OF P ( PART ) LOOP
    WHILE STATUS EQ 0  DO
    BEGINWHILE W44
        P <-  CURRENT
        GET PART : PART_P#,PART_COLOUR
        P_P# <-  PART_P#
        P_COLOUR <-  PART_COLOUR
        COMMENT START OF Q37 CODE
        COMMENT TUPLE COUNT FOR Q37
        N36 <-  0
        Q37 <-  FALSE
        FIND FIRST SUPPLY RECORD OF IS-SUPPLIED-BY SET
        SUPPRESS PARTAREA CURRENCY UPDATES .
        COMMENT START OF SP ( SUPPLY ) LOOP
        WHILE STATUS EQ 0  DO
        BEGINWHILE W45
            SP <-  CURRENT
            FIND OWNER RECORD OF SUPPLIES SET  SUPPRESS
            SUPPLIER-SET,SUPPAREA CURRENCY UPDATES .
            IF CURRENT EQ S  THEN
            BEGINIF I41
            N36 <-  N36 + 1
            Q37 <-  TRUE
            EXITWHILE W45
            ENDIF I41
            FIND NEXT SUPPLY RECORD OF IS-SUPPLIED-BY SET
            SUPPRESS PARTAREA CURRENCY UPDATES .
        ENDWHILE W45
        COMMENT END OF Q37 CODE
        IF Q37 NE TRUE  THEN
        BEGINIF I42
        N35 <-  N35 + 1
        Q36 <-  TRUE
        EXITWHILE W44
        ENDIF 142
        FIND NEXT DUPLICATE WITHIN PARTSET SET USING
        PART_COLOUR   SUPPRESS IS-SUPPLIED-BY,PARTAREA
        CURRENCY UPDATES .
    ENDWHILE W44
    COMMENT END OF Q36 CODE
    IF Q36 NE TRUE  THEN
    BEGINIF 143
    N34 <-  N34 + 1
    OUTPUT S_S#
    ENDIF 143
    FIND NEXT SUPPLIER RECORD OF SUPPLIER-SET SET  SUPPRESS
    SUPPLIES,SUPPAREA CURRENCY UPDATES .
ENDWHILE W43
COMMENT END OF Q35 CODE
838 MILLISECONDS
```

```
INPUT ALF STATEMENT
COMMENT
COMMENT DEFINE A VIEW CONTAINING THE COMPANIES WITH AVERAGE
COMMENT DEPARTMENTAL BUDGETS GREATER THAN 100000.
COMMENT
         DEFINE VIEW BIGC FROM C RANGE COMP,D RANGE DEPT
         USING C.C#,C.CNAME,C.CLOC
         WHERE AVG(D.BUDGET : C.C#=D.C#) GT 100000.
INPUT ALF STATEMENT
COMMENT
COMMENT DEFINE A VIEW IN TERMS OF THE VIEW BIGC,
COMMENT CONTAINING ALL THE BIGC TUPLES IN THE ACT.
COMMENT
         DEFINE VIEW BIGACT FROM Z RANGE BIGC USING Z.C#,Z.CNAME
         WHERE Z.CLOC="ACT".
INPUT ALF STATEMENT
COMMENT
COMMENT DO A RETRIEVAL ON THE VIEW (VIRTUAL RELATION) BIGACT
COMMENT
         RANGE OF X IS BIGACT.
INPUT ALF STATEMENT
COMMENT
         OUTPUT X.C#,X.CNAME  WHERE X.C# NE 123.
482 MILLISECONDS
START OF CODE-A
COMMENT START OF Q41 CODE
COMMENT TUPLE COUNT FOR Q41
N37 <-  0
COMP_CLOC <-  "ACT"
FIND COMP VIA CURRENT OF COMPSET USING COMP_CLOC   SUPPRESS
SS1,SS2,S1,COMPAREA CURRENCY UPDATES .
COMMENT START OF NEWV7 ( COMP ) LOOP
WHILE STATUS EQ 0  DO
BEGINWHILE W46
    NEWV7 <-  CURRENT
    GET COMP : COMP_C#,COMP_CNAME,COMP_CLOC
    NEWV7_C# <-  COMP_C#
    NEWV7_CNAME <-  COMP_CNAME
    NEWV7_CLOC <-  COMP_CLOC
    IF NEWV7_C# NE 123  THEN
    BEGINIF I44
    COMMENT START OF NEWQ3 CODE
    COMMENT TUPLE COUNT FOR NEWQ3
    N38 <-  0
    NEWQ3 <-  0
    FIND FIRST DEPT RECORD OF S1 SET  SUPPRESS S2,S3,
    DEPAREA CURRENCY UPDATES .
    COMMENT START OF NEWV9 ( DEPT ) LOOP
    WHILE STATUS EQ 0  DO
    BEGINWHILE W47
        NEWV9 <-  CURRENT
        GET DEPT : DEPT_BUDGET
        NEWV9_BUDGET <-  DEPT_BUDGET
        N38 <-  N38 + 1
        NEWQ3 <-  NEWQ3 + NEWV9_BUDGET
        FIND NEXT DEPT RECORD OF S1 SET  SUPPRESS S2,S3,
        DEPAREA CURRENCY UPDATES .
    ENDWHILE W47
    IF N38 NE 0  THEN
    BEGINIF I45
    NEWQ3 <-  NEWQ3 / N38
```

```
        ENDIF 145
        COMMENT END OF NEWQ3 CODE
        IF NEWQ3 GT 100000  THEN
        BEGINIF 146
        N37 <- N37 + 1
        OUTPUT NEWV7_C#,NEWV7_CNAME
        ENDIF 146
        ENDIF 144
        FIND NEXT DUPLICATE WITHIN COMPSET SET USING COMP_CLOC
        SUPPRESS SS1,SS2,S1,COMPAREA CURRENCY UPDATES .
    ENDWHILE W46
    COMMENT END OF Q41 CODE
    663 MILLISECONDS
```

```
INPUT ALF STATEMENT
COMMENT
COMMENT ANOTHER VIEW DEFINITION
COMMENT
      DEFINE VIEW DP
      FROM D RANGE DEPT,P RANGE PROJECT,RR RANGE PROJDEPT
      USING D.D#,D.C#,P.PROJ#,D.BUDGET RENAMED DBUDGET,
          P.BUDGET RENAMED PBUDGET
      WHERE
          D.D# = RR.D#
      AND D.C#=RR.C#
      AND P.PROJ#=RR.PROJ#
      AND P.BUDGET LT 100000
      AND D.DLOC ="ACT".
INPUT ALF STATEMENT
COMMENT
      RANGE OF X IS DP.
INPUT ALF STATEMENT
      OUTPUT X.C#,X.D#,X.PBUDGET WHERE X.DBUDGET GT 100000.
754 MILLISECONDS
START OF CODE-A
COMMENT START OF Q43 CODE
COMMENT TUPLE COUNT FOR Q43
N39 <-  0
FIND FIRST COMP RECORD OF COMPSET SET  SUPPRESS SS1,SS2,
COMPAREA CURRENCY UPDATES .
COMMENT START OF R32 ( COMP ) LOOP
WHILE STATUS EQ 0  DO
BEGINWHILE W48
    R32 <-  CURRENT
    GET COMP : COMP_C#
    R32_C# <-  COMP_C#
    DEPT_DLOC <-  "ACT"
    FIND DEPT VIA CURRENT OF S1 USING DEPT_DLOC   SUPPRESS
    S2,DEPAREA CURRENCY UPDATES .
    COMMENT START OF NEWV17 ( DEPT ) LOOP
    WHILE STATUS EQ 0  DO
    BEGINWHILE W49
        NEWV17 <-  CURRENT
        GET DEPT : DEPT_D#,DEPT_DLOC,DEPT_BUDGET
        NEWV17_D# <-  DEPT_D#
        NEWV17_DLOC <-  DEPT_DLOC
        NEWV17_BUDGET <-  DEPT_BUDGET
        IF NEWV17_BUDGET GT 100000  THEN
        BEGINIF I47
        FIND FIRST PROJDEPT RECORD OF S3 SET  SUPPRESS
        PARTAREA CURRENCY UPDATES .
        COMMENT START OF NEWV18 ( PROJDEPT ) LOOP
        WHILE STATUS EQ 0  DO
        BEGINWHILE W50
            NEWV18 <-  CURRENT
            FIND OWNER RECORD OF S4 SET  SUPPRESS PARTAREA
            CURRENCY UPDATES .
            NEWV16 <-  CURRENT
            GET PROJECT : PROJECT_BUDGET
            NEWV16_BUDGET <-  PROJECT_BUDGET
            IF NEWV16_BUDGET LT 100000  THEN
            BEGINIF I48
            N39 <-  N39 + 1
            OUTPUT R32_C#,NEWV17_D#,NEWV16_BUDGET
            ENDIF I48
```

```
              FIND NEXT PROJDEPT RECORD OF S3 SET  SUPPRESS
              PARTAREA CURRENCY UPDATES .
          ENDWHILE W50
          ENDIF I47
          FIND NEXT DUPLICATE WITHIN S1 SET USING DEPT_DLOC
          SUPPRESS S2,DEPAREA CURRENCY UPDATES .
      ENDWHILE W49
      FIND NEXT COMP RECORD OF COMPSET SET  SUPPRESS SS1,SS2,
      COMPAREA CURRENCY UPDATES .
  ENDWHILE W48
  COMMENT END OF Q43 CODE
  956 MILLISECONDS
```

```
INPUT ALF STATEMENT
QUIT

????? KILLED: GOODBYE

GARBAGE COLLECTIONS: 20 3
```