Raymond A. Lorie
Jorgen F. Nilsson

An Access Specification Language for a Relational Data Base System

We address the problem of executing high level language queries submitted to a relational data base system. As a step in the process of constructing an "efficient" compiler for a high level language we suggest the elaboration of an intermediate level language acting as a target language for the optimizer part of the compiler. This language may be conceived as one of several levels in a chain of abstract machines mapping a nonprocedural relational language onto primitive data base access operations. It is our conjecture that the introduction of an access specification language provides a conceptual platform facilitating the handling of the "optimization" problem.

Introduction

Several proposals for nonprocedural relational data base query languages have emerged in the past years, *e.g.* [1-3]. Besides the simple underlying data model the main advantage claimed by the creators of these languages is the fact that they are nonprocedural, since the access paths to be utilized in the eventual execution of the query are not specified explicitly in the language.

Evidently, the absence of access path specification places a heavy burden on the implementors of a compiler for such a language. The compilation algorithm must take into account the characteristics of the various access paths existing in the data base. Ideally, the compiler should act as an automatic programmer who, according to the physical characteristics of the data base, chooses an overall access strategy to be expressed as a procedural retrieval specification.

In this paper, however, we are not dealing with the optimization problem as such. We present a proposal for an intermediate language for a relational data base system. This language, called the *Access Specification Language* (ASL), allows for explicit and complete specification of the access paths to be exploited in the computation of a query. Thus, the language is procedural, yet it allows for the full specification of accesses without introduction of unnecessary details.

Since the language is intended to serve as an intermediate language in the compilation of relational data base expressions, ASL statements are represented by tree-like structures. However, for expository reasons we present an equivalent string language. There exists a close resemblance between the syntax classes of the string language and the nodes of the tree. In order to avoid confusions in naming, the procedures of the string language communicate among themselves via named arguments and return variables, much as do subroutines in a classical programming language. The names of the arguments and return variables are thus local names, whereas names of data base objects (stored in the catalog) are global. In the tree language the explicit data transfer amongst procedures is carried out by a simple addressing scheme (binding).

This work was done in the context of developing a compiler for the relational language SQL [1] for System R [4]. System R is a prototype developed at the IBM Research

Laboratory in San Jose, CA; it is intended only as a vehicle for data base research.

System R has two main components. The lower level component is the *Research Storage System* (RSS) with its corresponding interface (RSI). The upper level component, called *Relational Data System* (RDS), maps a high level SQL statement onto the RSS.

The issue of compilation *versus* interpretation has been discussed in [5]. The role of the ASL would be the same in a compiler or interpreter environment. Since a compilation approach is used in System R, the RDS functions are generally performed at compile time; the compiled code for each data base statement is in fact a series of calls to the RSI.

We shall first describe the target language of the compiler (the RSI). Then the structure of the RDS will be shown in order to place the ASL interface in its context. The Access Specification Language will then be defined. A first section will deal with queries involving a single relation. The following sections will then show how several ASL procedures are used together in the case of multirelation and/or multiblock queries. Finally a last section will describe how ASL is used to specify the processing needed for other SQL facilities like GROUP BY and HAVING.

Outline of Research Storage System (RSS)

This section describes the basic concepts of the RSS without any implementation considerations. Minor deviations have been introduced in order to simplify the description. Furthermore, aspects of the access method that are not essential to the purpose of this paper have been omitted. Thus, the notions of transactions, locks and segments, however important, are not covered.

- *RSS objects*

The RSS objects are:

1. *Relations* and *lists* constituting *tables* of data.
   A *table* is a sequence of *tuples*. Each tuple is an ordered sequence of fields, each field containing a single data value. The values of a field from all tuples of a table constitute a domain. When there is no ambiguity we do not make the distinction between field and domain. Relations and lists differ in the access operations as explained below.
2. Two types of access structures for relations:
   a. The *images* (indexes) give the illusion that the tuples of the relational table are sorted on the image key domain(s).
   b. The *binary links* make associations between tuples in two relations. A link is a set of disjoint link-occurrences, each linking one tuple from the parent relation to an arbitrary number of tuples from the child relation.
3. *Scans.*
   The scans support the access of relations and lists on a tuple-by-tuple basis. A scan is first open and then a NEXT operation is invoked to return the next tuple in the scan.

An RSS data base is a collection of named relations, lists, images and binary links with which are associated identifiers called respectively RID, LID, IID and BLID.

Relations, lists, links and images may be created and dropped by programs accessing the data base. The catalog part of the RSS data base (which we are not going to describe) contains the attributes of these objects. By contrast, the scans used for accessing the tuples are associated with the accessing programs and are therefore temporary. A scan (identified by a scan identifier, SID) is like an incarnation of a coroutine, offering a set of operations corresponding to the type of the scan. The scans come in four types: relational scans, list scans, image scans and binary link scans.

The order of the tuples retrieved through a relational scan is arbitrary (dependent on the physical order, which is subject to change). In a list the order of the tuples in the retrieval is that of the original insertion. A special function allows for the production of a sorted list. The order of the tuples in a link occurrence is the parent followed by the children in arbitrary but invariant order.

With each tuple in a given relation is associated a unique address called the tuple identifier (TID). RSS provides operations for the direct addressing of a tuple using the TID. In this context, however, we are dealing with TID's solely as an internal concept in scan operations.

- *RSS operations*

The *scan operations* are OPEN, NEXT, PARENT (binary link scan only), CLOSE.

OPEN:

The OPEN operation creates a scan and returns the SID of the scan.

Parameters:

Relational scan: RID

List scan: LID

Image scan: RID, IID (the identifier of an image on the relation RID) and a list of starting key values for the key domains in the image together with a match cri-

terion for the key values =, >= or >. A special match criterion, *first*, implies complete scan (key values ignored, hence).

B-link scan: BLID, SID. The SID must identify an open scan on one of the two relations associated with the binary link, thus providing a TID identifying the link-occurrence to be scanned. The scan is positioned before the first child tuple.

A scan object, once created, comprises internally an identifier of the data base object from which tuples are to be extracted (RID, LID, BLID on RID's, IID on RID) and a position on the object. In addition an image scan comprises a start key value used as starting point for the retrieval through the NEXT operation. At the opening, the scan is positioned before the first tuple of the object. The NEXT operation advances the scan to the next tuple of the object which fulfills the search predicates (see below) and returns the tuple value. In a sequence of NEXT operations on a scan, a tuple of the associated data object is visited at most once. Several scans may coexist on the same data base object.

NEXT:

The NEXT operation repositions a scan and returns a tuple from the relation or list associated with the scan (or a null value indicating that the scan has been exhausted).

Parameters: SID, a search predicate and some scan type dependent parameters described below. The search predicate is a normalized predicate (disjunctive normal form) composed of atomic comparison formulae:

```
search-predicate  ::= clause { OR clause }
clause            ::= comparison { AND comparison }
comparison        ::= DID relat-op constant
relat-op          ::= < | <= | > | >= | = | ¬=
DID               ::= domain identifier
                      (unique inside relation)
```

(The symbols [ ] indicate an optional occurrence, { } indicate zero, one or more occurrences.) In addition, for the image-scan a list of stop key values may be provided together with a stop criterion: <, <=, =, none.

PARENT: (for binary links)

Parameters: SID, search-predicate

The scan is positioned on the parent tuple of the current link-occurrence and the tuple value is returned provided that the search predicate is true for that tuple.

CLOSE:

Parameter: SID.

The scan object SID is dropped; *i.e.*, no references involving SID can be made to the data base.

CREATE and DROP relations, lists, images and links.

INSERT tuple into relation or list.

DELETE tuple from relation.

BUILDLIST:

Parameters: SID, search-predicate, list-sort-spec.

Produces a list of sorted tuples from a collection of tuples indicated by the SID of an open scan and a search predicate. The fields of the source table to be included in the list are indicated together with the sort attribute: ascending or descending. In addition, if SID identifies an image scan, a key value list and a stop criterion (<, <=, =, none) may be provided. Duplicate tuples may optionally be removed (UNIQUE attribute). The operation returns the LID of the target list of the sort operation.

Outline of the Relational Data System

Figure 1 represents schematically the overall structure of the RDS. The analysis phases imply classical compiler transformations and need no comments. Prior to the selection of an access strategy in the optimization phase, a "normalization" is performed on the internal representation of the query, involving integration of views and synonyms and conversion of some queries containing subqueries into queries containing a join but no subquery. In principle, the next phase of the optimization process takes into consideration potentially all ASL programs which would yield the answer to the query. The one estimated to be the most efficient in terms of CPU time and data base access operations is selected for synthesis. In the synthesis phase ASL is used to specify the procedure to be followed in order to produce the answer to the query.

Access Specification Language

ASL is introduced in a stepwise manner, starting with simple SQL examples and correspondingly simple ASL programs, and ending up with ASL constructs covering essentially all of SQL.

Syntactically an ASL program (in the string form) consists of procedures producing the query table result through mutual calls.

ASL-program ::= ASL-proc { ASL-proc }

The ASL procedures of a program communicate solely by explicit references (generally via a CALL) to a proce-

dure name and parameter/result transfer, therefore providing a decomposition of the overall task into rather independent units.

Single relation queries

• *Scan procedures*

A scan procedure is provisionally defined as follows:

```
scanproc ::=
  SCANPROC scanprocid [ ( param {, param } ) ];
    { temp-obj-defn ; }
    SCAN scanspec [ WHERE predicate ];
    [ IF restriction THEN ]
    RETURN ( ret-expr {, ret-expr } );
    END;
```

Initially the scan managed by the procedure is inactive. When the procedure is invoked for the first time the parameters are transferred, and the temporary objects that are defined in the procedure are evaluated. Thereafter, a scan is opened on a permanent data base relation or on one of the temporary objects just evaluated and a tuple value is returned. Every time the procedure is reinvoked, a tuple value is computed and returned until the scan is exhausted. The scan is then dropped.

Let us now analyze the various statements inside a scan procedure.

```
temp-obj-defn ::=
  LET RELATION relatid ( domid {, domid } ) =
                    buildprocid [ ( id {, id } ) ] |
  LET LIST listid    ( id {, id } ) =
                    buildprocid [ ( id {, id } ) ] |
  CREATE IMAGE imageid ON relatid FOR domid |
  LET ( id {, id } ) = buildvalueprocid [ ( id {, id } ) ]
```

The statements create respectively a relation, list, image, or scalar(s). These objects may be referred to in the procedure in which they are embodied using the established names (relatid and domids in the LET statement, for example). The temporary objects are dropped when the incarnation of the procedure is terminated (*i.e.*, at the exhaustion of the scan). For the creation of the relation or list the tuples to be inserted in the temporary objects are defined in a build procedure identified by buildprocid (see section "Build procedures"). Scalar values are defined by invoking a build value procedure; such scalars are used in the implementation of subqueries (see section "Sub-queries").

```
scanspec ::=
  relatid { image-spec | link-spec } | listid
```

A scan specification specifies the object to be scanned and the type of scan: relational-, image-, link- or list-scan.

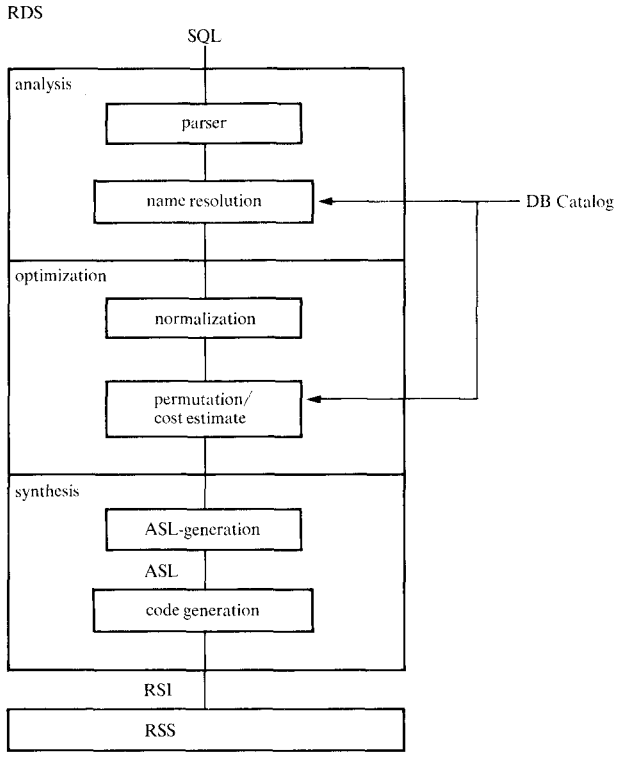

Figure 1 Structure of the *Relational Data System* (RDS).

```
image-spec ::=
  USING IMAGE imageid [ FROM constant ] [ TO constant ] |
  USING IMAGE imageid AT constant {, constant }
link-spec ::=
  USING LINK linkid (PARENT | CHLDRN) OF scanprocid
```

The link-occurrence to be scanned is specified implicitly as the scan position of the newest incarnation of the indicated scan procedure.

The WHERE clause is used to introduce the search predicate, which can be mapped directly onto the RSS search predicate. The logical restriction of the IF statement is discussed in section "Sub-queries." Note, however, that this restriction predicate allows for the specification of predicates which cannot be expressed by an RSS search predicate. It should also be noted that any time a constant is shown in the syntax it can also be a value passed as a parameter.

```
ret-expr   ::= arith-expr | aggr-expr
arith-expr ::= arithmetic expression
aggr-expr  ::= aggr-fct-id ( arith-expr ) | COUNT
```

The aggr-fct-id's identify aggregate functions like average, maximum, minimum and sum. COUNT is an argumentless function returning the number of tuples in the scan.

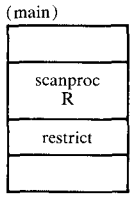

Figure 2 Scan procedure on relation R.

It is convenient to use a simple graphical representation to show the structure of the main ASL constructs, omitting all details. Figure 2 represents a scan procedure: The upper box represents the definition of temporary objects while the following ones represent the scan and the restriction part of the procedure. The lower box is used for a construct which will be explained later. The tag (main) indicates that this scan is seen by the caller of the ASL interface.

In the examples given throughout the paper, most of which are taken from [1], we use the following simple data base:

```
EMP (EMPNO, ENAME, SAL, DNO, JOB, AGE, MGR)
DEPT (DNO, DNAME, LOC)
```

Example 1

```
SELECT NAME
FROM EMP
WHERE DNO = 50
```

If a relation scan is used in connection with a search predicate the ASL program becomes

```
SCANPROC A;
  SCAN EMP WHERE DNO = 50;
  RETURN (NAME);
  END;
```

This procedure (imagine it is called from the environment) specifies a relational scan to be opened on EMP using the search predicate DNO = 50. The scan procedure A performs an RSS NEXT operation every time it is invoked and returns the NAME field of each tuple.

Utilizing an image on DNO the program becomes

```
SCANPROC A;
  SCAN EMP
    USING IMAGE EMP_DNO AT 50;
  RETURN (NAME);
  END;
```

Such a procedure can be easily compiled as shown in [5]. The next paragraph shows the object program in a PL/I-like language. STATUS is a variable permanently associated with the scan procedure. It is initialized to 'INACTIVE'.

```
PROC A;
  DCL STATUS STATIC INIT('INACTIVE');
  IF STATUS = 'INACTIVE' THEN
    DO; sid = OPEN-REL-SCAN (EMP);
        STATUS = 'ACTIVE';
    END;
  tuple = NEXT(sid, <DNO, =, 50>);
  IF tuple = NULL THEN
    DO; CLOSE (sid);
        STATUS = 'INACTIVE';
        RETURN (NULL);
    END;
  RETURN (tuple.NAME);
  END;
```

Example 2

```
SELECT NAME, DNO
FROM EMP
ORDER BY DEPT
```

If there exists an image on the column DNO of the relation EMP the following ASL procedure could be used:

```
SCANPROC A;
  SCAN EMP
    USING IMAGE EMP_DNO;
  RETURN (NAME,DNO);
  END;
```

If such an image does not exist one needs to sort the result and the scan will be done on the temporary list containing the sorted tuples. The following construct will be used to that effect.

- *Build procedures*

```
buildproc ::=
  BUILDPROC buildprocid [ ( param {, param } ) ];
    { temp-obj-defn ; }
    SCAN scanspec [ WHERE search-predicate ];
    [ IF restriction THEN ]
    INSERT INTO ( RELAT | LIST ):
        id {, id } [ SORTED BY id {, id } [ UNIQUE ] ];
    END;
```

Syntactically the build procedure is like the scan procedure except for the INSERT clause. The semantic difference is that the build procedure does not return individual tuples during a scan, but rather accumulates the whole result in an object, the RSS identifier of which is returned to the invoker. The INSERT statement specifies the type of the object to be created (relation or list) and the names to be given to its columns.

Figure 3 shows the graphical representation of a scan procedure invoking a build procedure producing the list L. Note the meaning of the arrow originating in the upper box of the scan procedure: Before opening the scan on the list L the list is constructed by invoking the build procedure. This procedure is not seen by the caller of the ASL interface and is tagged (sub) by analogy to subroutines.

The following example shows how a build procedure is used for the compilation of an SQL query with the ORDER BY option.

Consider again example 2. Tuples are ordered by producing a temporary list of sorted tuples. Since the program is supposed to deliver the result on a tuple-by-tuple basis the list is scanned by the (main) procedure A.

```
SCANPROC A;
  LET LIST L(NM,DEP) = B;
  SCAN L;
  RETURN (NM,DEP);
  END;
BUILDPROC B;
  SCAN EMP;
  INSERT INTO LIST: NAME,DNO SORTED BY DNO;
  END;
```

The LET statement in procedure A invokes B, which returns the RSS lid of the list created in B. The list is assigned the local name L with domains NM,DEP for reference in A.

The compilation would produce the following program:

```
PROC A;
  DCL STATUS STATIC INIT('INACTIVE');
  IF STATUS = 'INACTIVE' THEN
    DO; lid = B;
        sid = OPEN-LIST-SCAN(lid);
        STATUS = 'ACTIVE';
    END;
  tuple = NEXT (sid);
  IF tuple = NULL THEN
    DO; CLOSE(sid);
        STATUS = 'INACTIVE';
    END;
  RETURN (tuple);
  END;
PROC B;
  sid = OPEN-REL-SCAN(EMP);
  lid = BUILDLIST(sid,NM,DNO,increasing DNO);
  RETURN(lid);
  END;
```

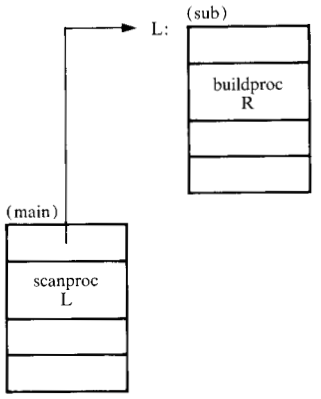

Figure 3 Scan and build procedures.

Collaborating ASL procedures

We have introduced the two basic building blocks of the ASL and we now turn to the more general case of queries involving more than one relation.

Basically, the FROM clause of a SELECT statement in SQL specifies a Cartesian product subject to a restriction in the WHERE clause. (Frequently these restrictions are such that the Cartesian product becomes an equi-join.)

Interaction among ASL procedures for the specification of joins is specified via a new ASL statement: FOREACH TUPLE. The definition of the scanproc (and buildproc) is completed as follows:

```
scanproc ::=
  SCANPROC . . .
    . . .
    [ IF restriction . . . ]
    FOREACH TUPLE [ id {, id } UNIQUE ];
      [ LET id {, id } = scanprocid [ ( id {, id } ) ]; ]
    RETURN . . .
    END;
```

Suppose a first scan procedure, say S, retrieves, upon invocation, a tuple, say t, fulfilling the search predicate of the WHERE clause and the restriction of the IF clause. If the FOREACH TUPLE statement is present, the LET statement specifies a scan procedure, say S′, which is itself invoked with parameter values corresponding to the current tuple t. The scan procedure S′ returns a tuple t'. The fields of both t and t' may participate in the return expressions specified in the RETURN clause of S. A subsequent invocation of S will not advance the scan defined inside S but will advance the scan inside S′ and return a

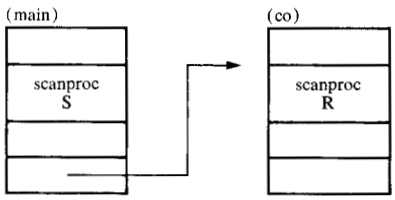

Figure 4 Join using FOREACH TUPLE.

new tuple t'' yielding a composite tuple t, t''. Only when the scan in S′ is exhausted is the scan in S advanced. For each new tuple in S the scan procedure S′ is reinvoked with new parameter values and the temporary objects are then reevaluated. The UNIQUE attribute can be used only if the scan is done along an access path (list or image) which insures that tuples are ordered on the columns specified with the UNIQUE attribute. (This feature is used in queries involving GROUP BY.)

Figure 4 represents graphically such a join mechanism. (co) is used by analogy to coroutines.

The principle is illustrated below for an equi-join between two relations.

Example 3

```
SELECT ENAME, DNAME
FROM EMP, DEPT
WHERE EMP.DNO = DEPT.DNO
    AND LOC = 'Evanston'
```

Suppose there exists an index on the LOC domain of DEPT and also an index on the DNO domain of the EMP relation. The optimizer may have decided to rearrange the (virtual) Cartesian product generation, exploiting the EMP relation through the DEPT relation:

```
SCANPROC A;
  SCAN DEPT USING IMAGE DEPT_LOC AT 'Evanston';
  FOREACH TUPLE
    LET ENAME = B(DNO);
  RETURN(ENAME,DNAME);
  END;
SCANPROC B(DNUM);
  SCAN EMP USING IMAGE EMP_DNO AT DNUM;
  RETURN(ENAME);
  END;
```

It is essential to note that for each tuple found by scanproc A the scanproc B is activated as often as needed to exhaust its scan. Only then is the scan in A advanced. Also, every scanproc considers only the tuples that satisfy the predicates local to the corresponding scan, thereby avoiding the development of a full Cartesian product when only a small portion of it is of any interest.

The object programs for such a pair of procedures appear as follows:

```
PROC A;
  DCL STATUS STATIC INIT('INACTIVE');
  IF STATUS = 'INACTIVE' THEN
    DO; sid = OPEN-IMAGE-SCAN(DEPT,DNO,'Evanston');
        STATUS = 'ACTIVE';
     X: tuple = NEXT(sid);
        IF tuple = NULL THEN
          DO; CLOSE(sid);
              STATUS = 'INACTIVE';
          END;
    END;
  tuple' = B(tuple.DNO);
  IF tuple' = NULL THEN GO TO X;
  RETURN (tuple'.ENAME, tuple.DNAME));
  END;
PROC B(DNUM);
  DCL STATUS STATIC INIT('INACTIVE');
  IF STATUS = 'INACTIVE' THEN
    DO; sid = OPEN-IMAGE-SCAN(EMP,DNO,DNUM);
        STATUS = 'ACTIVE';
    END;
  tuple = NEXT(sid);
  IF tuple = NULL then
    DO; CLOSE(sid);
        STATUS = 'INACTIVE';
    END;
  RETURN(tuple);
  END;
```

Note that the ASL definition for procedure B specifies that the index be used. It could also have specified that a link to be used (if there is a link from DEPT to EMP). The scan procedure B would then become

```
SCANPROC B';
  SCAN EMP USING LINK DEPT_EMP CHILDRN OF A;
  RETURN(NAME);
  END;
```

The explicit reference to the procedure name A in B′ above is not an invocation of A; it only indicates the link occurrence on which the link scan in B′ is to be opened.

General joins

Having shown how simple SQL queries can be expressed in terms of ASL programs and how the latter programs can be compiled into RSS operations, we now take up the problems associated with the compilation of SQL into ASL at a more abstract and general level.

Rather than making an account of a complete compilation algorithm we hint at some underlying principles in an optimizing compiler, providing at the same time the background and justification for the introduction of particular linguistic notions in ASL.

Conceptually, an SQL query is decomposed into (sub)query units, called SELECT-blocks, each of which consists of the SELECT and FROM clause, possibly with a WHERE, GROUP BY, HAVING or ORDER BY clause.

The process for computing general joins can be described by using the scan procedures and build procedures introduced in the previous section. Let us first look at a case in which no intermediate result needs to be created.

For each SELECT block participating in a query the optimizer has to decide as to the order in which the virtual Cartesian product of the participating relations must be built (see example 3). Likewise the optimizer has to select among the various access paths such as images *versus* links for the individual relations of the chosen permutation.

Assuming that the optimizer part of the compiler is able to carry out this task utilizing suitable cost estimate functions, the generation of ASL programs is, in principle, straightforward: Let $R_1, R_2, \cdots, R_n$ be the original sequence of relations in a SELECT-block and $R_{i1}, R_{i2}, \cdots, R_{in}$ the permutation chosen in the cost evaluation process. The ASL specification consists of a chain of scan procedures, each procedure invoking the next one on the chain. The relation R_{i1} is scanned. For each tuple retrieved relation R_{i2} is scanned. For each composite tuple R_{i1}, R_{i2} the relation R_{i3} is scanned etc. This generalizes the concept of collaborating ASL procedures. At any point in the process one can consider that we are joining a virtual *composite scan* (for example $R_{i1}, R_{i2}, \cdots, R_{ij}$) with the next scan $R_{i(j+1)}$. To simplify the notation let us rename $R_{i1}, R_{i2} \cdots$ as $R_1, R_2 \cdots$.

As far as the Boolean expression of the query WHERE clause is concerned, individual conjuncts are to be distributed along the chain of scan procedures so that each conjunct contains references to data obtained in its own scan procedure or in any scan procedure to the left.

Some of the predicates may be integrated into and exploited directly in RSS operation(s); this is often the case for equi-join predicates when links and images can be used. The remaining distributed predicates of the original WHERE clause form the restriction clause of the ASL procedures.

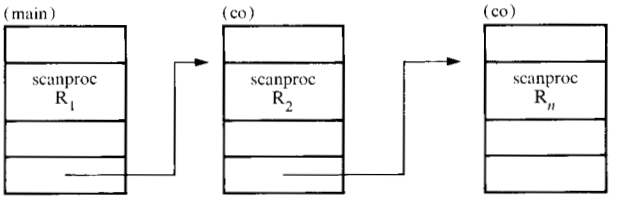

Figure 5 Join specification.

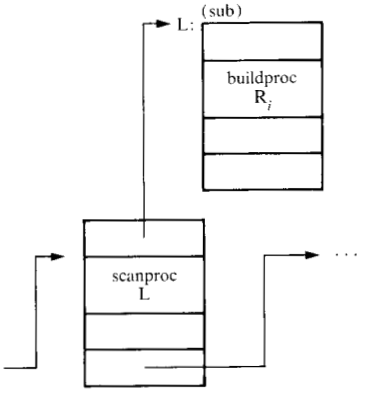

Figure 6 Using a temporary relation.

This decomposition scheme applies recursively to SELECT blocks participating in Boolean expressions as subqueries of the major query. Hence, restriction clauses may comprise references to a chain of ASL procedures, as we shall see later when explaining the restriction clause in detail.

General joins can sometimes be produced more efficiently if intermediate results are created. In such cases build procedures are used in connection with scan procedures. The generation of temporary objects is potentially worthwhile (the cost estimator has to decide) in two situations.

a. The cardinality of one of the factor relations, say R_i, can be diminished using a local predicate, *i.e.*, a predicate referring to columns of the relation R_i only. A build procedure is first used to create a temporary result (a list, for example) and then a scan procedure is used to scan the temporary object. If such a method is used for R_1, Fig. 5 is altered as shown in Fig. 6.
b. It may be advantageous to compute a join by using a mechanism resembling a merge of ordered lists. Assume both relations are ordered on the join domains. Then the first relation is scanned normally. For every tuple returned by the first scan the group of matching tuples in the second relation is scanned (the tuples in

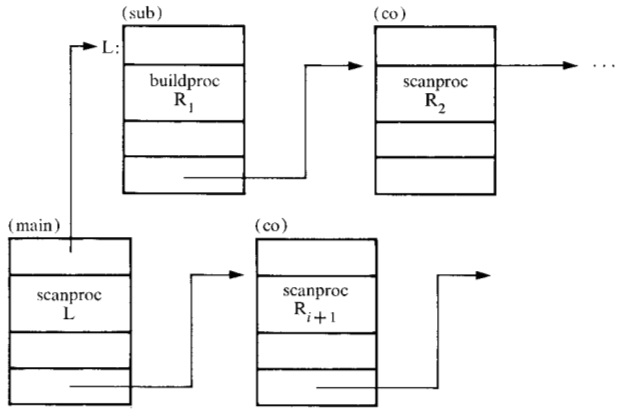

Figure 7 Intermediate result for a join.

the group are contiguous because of the ordering). The top position of a group is remembered so that the same group can be efficiently scanned again for the next tuples in the first scan if they happen to have the same value(s) for the join domain(s). Generally, lists must be built in order to provide the relevant orderings. For example, if we refer to Fig. 5, the method can use a merge to join the composite scan, say $(R_1, R_2, \cdots, R_i)$, with the relation R_{i+1}. This can be done only if the composite scan returns the tuples in the appropriate order. If this is not the case the composite tuples should be materialized before the join with R_{i+1} can be performed. Again a build procedure is used to create the sorted intermediate result. This build procedure will itself invoke all scan procedures needed to obtain the composite tuples. Figure 7 illustrates this case.

Sub-queries

The restriction clause in the scan (or build) procedure has been introduced earlier. We now show the full power of the restriction clause in connection with the compilation of subqueries.

```
restriction  ::= ( restriction bool-op restriction )
                 | NEG restriction | pred-block
bool-op      ::= AND | OR
pred-block   ::= ( { temp-obj-defn ; } comparison )
comparison   ::= val-comp | set-comp
val-comp     ::= arith-expr comp-op arith-expr
comp-op      ::= < | > | <= | >= | = | ¬=
set-comp     ::= ident set-op ident
set-op       ::= SUBSET | PROPSUBSET | EQSET
ident        ::= relation-id | list-id
```

A predicate block (pred-block) in a particular scanproc S is used instead of a comparison only when one (or both) of the operands of the comparison is the result of a subquery the result of which depends on the tuple retrieved in this scanproc S (this case is called correlation in SQL). If the subquery is not correlated, then it can be evaluated as a definition at the top of the scanproc S, or even before the scanproc S is invoked.

Example 4

Consider the following example with a correlated subquery:

```
SELECT NAME
FROM EMP X
WHERE SAL > SELECT AVG(SAL)
            FROM EMP Y
            WHERE X.DNO = Y.DNO
```

In a straightforward computation of the above query a scan would be defined on the relation EMP. Then for each tuple returned by that scan another procedure would be invoked which should return as a scalar value the average salary in the department. In order to distinguish between a scan returning a single tuple as result of an aggregation function and a scalar, a new construct called build value procedure is introduced. It is syntactically similar to a scanproc retrieving a single tuple.

The ASL program would be as follows:

```
SCANPROC A;
  SCAN EMP
  IF (LET AVGSAL = B(DNO) ; SAL > AVGSAL) THEN
     RETURN(NAME)
  END;
BUILDVALUESCAN B(DNUM);
  SCAN EMP;
  IF DNO = DNUM THEN
      RETURN (AVG(SAL));
  END;
```

The BUILDVALUE procedure behaves exactly as a scan procedure except that it returns a value rather than a scan.

Figure 8 represents subqueries graphically: (a) illustrates a case without correlation: the value of the subquery can be evaluated once at the beginning of the query; (b) illustrates a case with correlation: the subquery is called from within the restriction clause and therefore evaluated once for each tuple in the scan.

Set comparison

Set comparison can occur in the restriction clause. The following shows how ASL handles these comparisons.

Example

Find those departments in which all job types are represented:

```
SELECT DNO
FROM EMP X
WHERE (SELECT UNIQUE JOB
       FROM EMP)
      =
      (SELECT UNIQUE JOB
       FROM EMP.Y
       WHERE Y.DNO = X.DNO)
```

A possible ASL specification is as follows:

```
SCANPROC A;
  LET LIST ALLJOBS(JOB) = B;
  SCAN EMP;
  IF (LET LIST DNOJOBS(JOB) = C(DNO);
            ALLJOBS EQSET DNOJOBS) THEN
    RETURN(DNO);
  END;
BUILDPROC B;
  SCAN EMP;
  INSERT INTO LIST: JOB SORTED BY JOB UNIQUE;
  END;
BUILDPROC C(DNUM);
  SCAN EMP WHERE DNO = DNUM;
    INSERT INTO LIST: JOB SORTED BY JOB UNIQUE;
  END;
```

The set comparison is carried out as a comparison of sorted lists, in a fashion similar to merging. Alternatively, one may choose to perform the set comparison by comparing appropriately indexed relations (using two parallel scans on the comparable key domains of the images):

```
SCANPROC A';
  CREATE IMAGE I1 ON EMP FOR JOB;
  SCAN EMP;
  IF (LET RELAT DNOJOBS(JOB) = B(DNO);
       CREATE IMAGE I2 ON DNOJOBS FOR JOB;
       I1 EQSET I2) THEN
  RETURN(DNO);
  END;
```

Queries with GROUP BY / HAVING clauses

GROUP BY queries lead to ASL programs very similar to those obtained for queries with subqueries (see example 4).

Example 5

```
SELECT DNO, AVG(SAL)
FROM EMP
GROUP BY DNO
```

leads to the following ASL program (note how duplicates are eliminated:

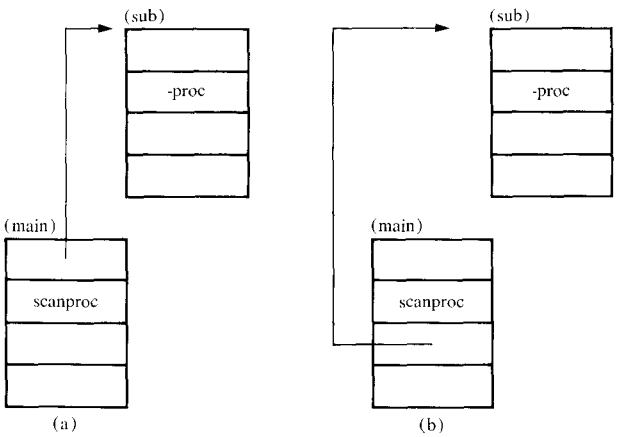

Figure 8 Specification of subqueries: (a) without correlation; (b) with correlation. ("-proc" stands for buildproc or buildvalue-proc.)

```
SCANPROC A;
  LET LIST L(DNO) = B;
  SCAN L;
  FOREACH TUPLE
    LET AVGSAL = C(DNO);
  RETURN (DNO, AVGSAL);
  END;
BUILDPROC B;
  SCAN EMP;
  INSERT INTO LIST: DNO UNIQUE;
  END;
SCANPROC C(D);
  SCAN EMP WHERE DNO = D;
  RETURN (AVG(SAL));
  END;
```

It is often efficient to compute GROUP BY queries by building a list of tuples sorted on the group column(s). Such a strategy uses the UNIQUE attribute of the FOREACH statement; it is expressed as follows:

```
SCANPROC A;
  LET LIST L(DNO,SAL) = B;
  SCAN L;
  FOREACH TUPLE DNO UNIQUE
    LET AVGSAL = C(DNO);
  RETURN (DNO, AVGSAL);
  END;
BUILDPROC B;
  SCAN EMP;
  INSERT INTO LIST: DNO,SAL SORTED BY DNO;
  END;
SCANPROC C(D);
  SCAN EMP WHERE DNO = D;
  RETURN (AVG(SAL));
  END;
```

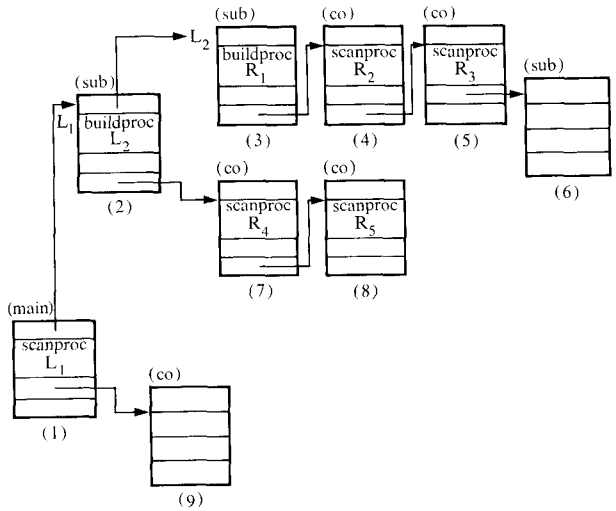

Figure 9 A complex ASL specification.

Also, the more general case of a combined GROUP BY / HAVING query can be handled with the constructs previously described. We give a generic example:

Example 6

```
SELECT f(COL1,COL2)
FROM R
WHERE pred1
GROUP BY COL1,COL2
HAVING pred2(aggr-func(COL3))
```

A possible ASL program is as follows:

```
SCANPROC A;
  LET LIST L(COL1,COL2) = B;
  SCAN L;
  IF (LET AGGRID = C(COL1,COL2);
        pred2(AGGRID) THEN
  RETURN(f(COL1,COL2));
  END;
BUILDPROC B;
  SCAN R;
  IF pred1 THEN
    INSERT INTO LIST: COL1,COL2
          ORDER BY COL1,COL2 UNIQUE;
  END;
BUILDVALUEPROC C(P1,P2);
  SCAN R;
  IF (COL1 = P1 AND COL2 = P2) THEN
    RETURN(aggr-func(COL3));
  END;
```

Building of ASL structures for complex queries

The constructs that have been presented can be used as building blocks to develop arbitrarily complex ASL specifications. Figure 9 gives schematically an example of such a complex specification.

Blocks 1 and 9 implement a GROUP BY as shown in example 5 (second strategy). In order to use that strategy a sorted list L_1 must be produced. This is done by a buildproc (2) which inserts in a list (and sorts) the tuples obtained by joining the composite tuples stored in L_2 with tuples from relation R_4 (block 7) and R_5 (block 8). L_2 is built by scanning R_1 (block 3), joining them with tuples in R_2 and R_3, using a complex join predicate involving a subquery correlated with values in R_1, R_2, R_3 (block 6).

Insert, delete, update

In the implementation of System R ASL has been extended to support not only queries but deletions, updates and insertions. The principle of these extensions is simple: The RETURN statement in the (main) scan procedure is replaced by a delete or update statement or by an insert statement which specifies that the returned tuples must be inserted into a previously defined relation. These extensions could also be used to implement set operations like union and difference.

Concluding remarks

A general scheme for the compilation of SQL into primitive data base operations has been established in the form of a language which can be used as target in the decomposition of arbitrarily complex queries. The decomposition has been illustrated by outlining the general principles and supporting them with examples.

It should be mentioned here that the idea of decomposition has been used in other systems. Let us discuss the approaches used in [3] and [6].

Consider, for example, the sample query used in [3] to illustrate the optimization in INGRES. The sample data base is similar to the one used throughout this paper and the query consists of finding the employees over 40 working at a given location and who make more than their managers. In [3] it is shown how such a query is decomposed into four simple ones, three of them using temporary files T_1, T_2 and T_3:

1. store in T_1 the departments at the given location;
2. store in T_2 the employees over 40;
3. for each value x of a department in T_1 construct a T_3 containing the employees in T_2 who work in department x;
4. for each tuple in T_3 use the EMP file to find out if his salary is larger than the salary of his manager.

Such a decomposition resembles the decomposition used in ASL. It is, however, more algebraic and has the disadvantage of requiring temporary files to be built at each step. In ASL the same strategy can be expressed—and we leave this as an exercise for the reader. But ASL allows many alternative strategies to be specified. The following one, for example, does not require any intermediate file:

1. fetch a department;
2. for each such department find the employees over 40 that work in that department;
3. for each such employee find the salary of the manager and compare it with the salary of the employee.

We are not saying that such a strategy is better but only showing the flexibility of ASL for expressing a wide variety of strategies. Note also that ASL allows for the complete specification of the access path to be used in the simple query.

In PRTV [6] the result of the optimizer is an algebraic expression with operators like projection, join, selection etc. Such an expression is then given as input to the interpreter. This approach is again algebraic and intermediate results are created. In ASL the strategies are described at a lower, more procedural level. This allows for describing how a join is to be computed rather than considering the join as a basic operator.

These comparisons emphasize the flexibility of the ASL approach. It should also be clear, for example, that it is possible to write an ASL program using a relation which does not appear in the query itself. Such a relation, if properly indexed, may provide an efficient access path into a relation of the query, *e.g.*, if the two relations are coupled by an appropriate link. We realize that certain limitations exist in the current form of ASL. For example, it does not allow for the specification of some join methods proposed in [7], based on the manipulation of lists of tuple identifiers. We believe, however, that our basic decomposition scheme can readily be extended to support some of these most sophisticated strategies.

Finally, although we have addressed the problem of compiling SQL 2 in particular, the decomposition principles outlined as well as the linguistic form of ASL are believed to be of general interest in the context of relational query languages.

Acknowledgments

The authors acknowledge the support of W. F. King and D. Chamberlin. Also we are grateful to M. Astrahan and P. Griffiths for their discussions and suggestions leading to clarification of some points throughout the preparation of this paper.

Appendix: ASL syntax

```
ASL-program ::= scanproc { scanproc | buildproc |
                           buildvalueproc }
scanproc ::=
  SCANPROC scanprocid [ ( param {, param } ) ] ;
    { temp-obj-defn ; }
    SCAN scanspec [ WHERE predicate ];
    [ IF restriction THEN ]
    FOREACH TUPLE [ id {, id } UNIQUE ];
      [ LET id {, id } = scanprocid [ ( id {, id } ) ]; ]
    RETURN ( ret-expr {, ret-expr } );
    END;
buildproc ::=
    BUILDPROC buildprocid [ ( param {, param } ) ];
    { temp-obj-defn ; }
    SCAN scanspec [ WHERE predicate ];
    [ IF restriction THEN ]
    FOREACH TUPLE [ id {, id } UNIQUE ];
      [ LET id {, id } = scanprocid [ ( id {, id } ) ]; ]
    INSERT INTO ( RELAT | LIST ):
        id {, id } [ SORTED BY id {, id } [ UNIQUE ] ];
    END;
buildvalueproc ::=
  BUILDVALUEPROC . . . same as SCANPROC . . .
temp-obj-defn ::=
  LET RELATION relatid ( domid {, domid } ) =
                        buildprocid [ ( id {, id } ) ] |
  LET LIST listid ( id {, id } ) =
                      buildprocid [ ( id {, id } ) ] |
  LET ( id {, id } ) =
            buildvalueprocid [ ( id {, id } ) ] |
  CREATE IMAGE imageid ON relatid FOR domid
scanspec ::=
  relatid { image-spec | link-spec } | listid
image-spec ::=
  USING IMAGE imageid [ FROM constant ] [ TO constant ] |
  USING IMAGE imageid AT constant {, constant }
link-spec ::=
  USING LINK linkid ( PARENT | CHLDRN ) OF scanprocid
predicate   ::= clause { OR clause }
clause      ::= comparison { AND comparison }
comparison  ::= DID relat-op constant
relat-op    ::= < | <= | > | >= | = | ¬=
DID         ::= domain identifier
restriction ::= ( restriction bool-op restriction )
                | NEG restriction | pred-block
bool-op     ::= AND | OR
pred-block  ::=( { temp-obj-defn ; } comp )
comp        ::= val-comp | set-comp
val-comp    ::= arith-expr comp-op arith-expr
```

comp-op ::= < | > | <= | >= | = ¬=
set-comp ::= ident set-op ident
set-op ::= SUBSET | PROPSUBSET | EQSET
ident ::= relation-id | list-id
ret-expr ::= arith-expr | aggr-expr
arith-expr ::= arithmetic expression
aggr-expr ::= aggr-fct-id (arith-expr) | COUNT

References

1. D. D. Chamberlin, M. M. Astrahan, K. P. Eswaran, P. P. Griffiths, R. A. Lorie, J. W. Mehl, P. Reisner, and B. W. Wade, "SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control," *IBM J. Res. Develop.* **20,** No. 6, 560-575 (1976).
2. M. M. Zloof, "Query by Example," *Proc. AFIPS 1975 NCC* **44,** AFIPS Press, Montvale, NJ, pp. 431-437.
3. M. Stonebraker, E. Wong, P. Kreps and G. Held, "The Design and Implementation of INGRES," *ACM Trans. Database Syst.* **1,** No. 3, 189-222 (1976).
4. M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson, "SYSTEM R: Relational Approach to Data Base Management," *ACM Trans. Database Syst.* **1,** No. 2, 97-137 (1976).
5. R. A. Lorie and B. W. Wade, "The Compilation of a Very High Level Data Language," *Research Report RJ2008*, IBM Research Laboratory, San Jose, CA, 1977.
6. S. J. P. Todd, "The Peterlee Relational Test Vehicle—a system overview," *IBM Syst. J.* **15,** No. 4, 285-308 (1976).
7. M. W. Blasgen and K. P. Eswaran, "On the Evaluation of Queries in a Relational Data Base System," *Research Report RJ1745*, IBM Research Laboratory, San Jose, CA, 1976.

Received October 25, 1978; revised December 21, 1978

Raymond A. Lorie is located at the IBM Research Division laboratory, 5600 Cottle Road, San Jose, California 95193; Jorgen F. Nilsson is at Grundtvicsvej 8B, 3. DK-1864 Copenhagen, Denmark.