F. Antonacci
P. Dell'Orco
V. N. Spadavecchia
A. Turtur

AQL: A Problem-solving Query Language for Relational Data Bases

Abstract: AQL is a query language based on a data base management system which uses Codd's relational model of data. It has been designed mainly to be used by the nonspecialist in data processing for interactive problem solving, application building, and simulation. Ease of use is achieved by providing an interface which allows the use of default options, synonyms, and definitions of attributes, inference, and the possibility of interactive completion of the query (i.e., menu). AQL combines the capabilities of the relational model of data with the powerful computational facilities and control structure of the host programming language (i.e., APL). A prototype version of AQL, which has been implemented, is reviewed.

Introduction

Data base management systems and their associated query languages have evolved rapidly since the introduction by Codd, in his 1970 historical paper [1], of the relational model of data. This model has been appealing to data base researchers both for the formal precision of its definition [2, 3] and for the simplicity, symmetry, and semantic completeness of the conceptual model of the data available to the final user. The user is presented with tables (called relations), which can be named and on which he can execute the well-known operations of table look-up and comparison. In fact, several query languages have been proposed based on Codd's data model, most of them explicitly intended for the inexperienced user. Examples include Query-by-Example [4], in which the user is required simply to describe an example of a possible answer to the query he has in mind, and SEQUEL [5], in which the query is formulated in a structured form with English keywords; similar approaches have been taken in SQUARE [6] and QUEL [7].

In designing AQL (an acronym for A Query Language), which is based on the *n*-ary relational model, a major objective was to have a query language and a system which could provide a "value added" in terms of level of intelligence or flexibility made available to the user, whether a computer specialist or not. Thus it was important that the language, even though formal, have not only a simple syntax (hence one easy to learn and to use), but also a high level of capability and effectiveness in problem solving and simulation, for the cases in which it was necessary not only to retrieve data but also to process them in an interactive and unanticipated way.

To achieve this objective, the following facilities have been introduced into the language:

1. Descriptiveness, so the user has only to describe *what* is to be retrieved, rather than *how* it is to be retrieved;
2. Richness of default options, so that "obvious" actions (for example, quoting constants, referring attributes to the proper relation, etc.) could be done automatically by the system;
3. The possibility of using synonyms and definitions of attribute names, which allows the user either to refer to a given attribute either by its name or by one of its synonyms or else to reference an attribute defined as a function of pre-existing ones;
4. An automatic recall of an inference mechanism, which gives to the system the facility of building for the user, whenever possible, all the necessary intermediate steps in queries requiring navigation through relations;
5. An interactive query completion facility, which is automatically activated in cases of incomplete queries, or whenever the application of default options or infer-

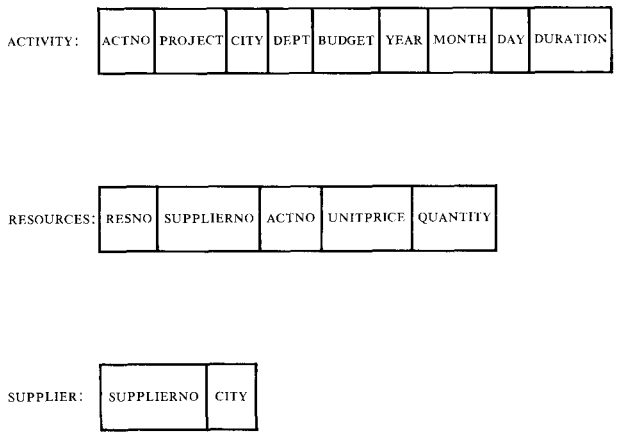

Figure 1 Data base PROJECTS.

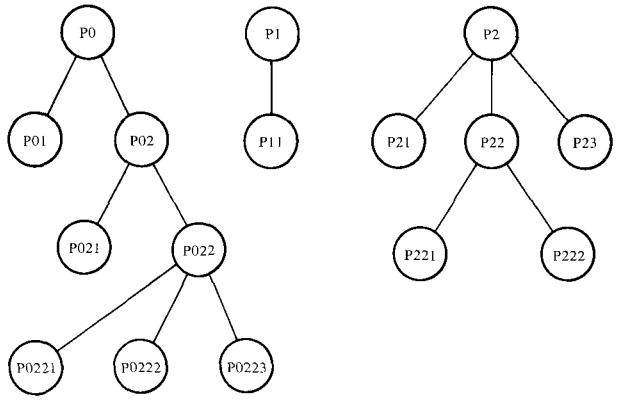

Figure 2 Structure of projects and activities.

ences may lead to ambiguities; in these cases the user is asked to give, by means of a menu, the lacking information;

6. Eventually, an open-endedness capability (which is built into the language) by which the user can define new functions of the language using the pre-existing ones, provided that he conforms to the language syntax.

To give the user the ability to execute unanticipated computations on retrieved data, the query language has been conceived as an extension of a programming language, so as to preserve a homogeneous environment in which queries and algorithms can be executed. This led us to the choice of APL [8–10] both as the implementation language and system for AQL and as host language to process the results of queries.

The syntax of AQL follows that of APL, so, strictly speaking, AQL is not a new language; rather it can be considered a super-system of APL. AQL has its own interpreter with a set of basic primitive functions made up by the primitives of the query language together with all the APL primitives. This approach results in complete "transparency" between the query language and the programming language. In fact, both elementary and user-defined functions may be used in the body of a query, and queries may be used in the body of user-defined functions; furthermore (as will be seen later), the values retrieved as results of queries are APL objects, which are made available in the workspace for further processing.

AQL is the language interface supported by a relational data base management system prototype developed at the IBM Bari Scientific Center. The general architecture of the system is presented in Appendix A. It provides facilities for data definition (e.g., creating and adding relations to the data base) and update of relations.

In this paper the characteristics of AQL are presented by a series of examples and by using the data base in Fig. 1. In this figure, the relation ACTIVITY contains information about projects; it is described by a set of attributes: ACTNO is the activity number; it is also the primary key of the relation as defined by Codd [2], and PROJECT gives the project number to which a particular activity belongs. The other attributes indicate the city, department, budget, starting date, and duration of each activity. A given project can itself be an activity, thus having another project upon which it is dependent; other projects may have no direct ancestors, thus being themselves the ancestors of all activities and/or projects dependent on them. Typical situations are represented in Fig. 2. For simplicity, let us suppose that the structure is strictly hierarchical; i.e., every activity has at most one predecessor.

The relation RESOURCES in Fig. 1 describes, for each resource number, the supplier, the activity that uses that particular resource, and the price for one unit of it together with the quantity of resources required by that activity. Obviously, an activity can have more than one supplier, whereas a given supplier can supply more than one activity with the same or different resources. In this case, the primary key of the relation is given by the three attributes (RESNO, SUPPLIERNO, ACTNO). The final relation in Fig. 1, SUPPLIER, gives, for each supplier, the city where he is located.

In the following sections, this data base is called PROJECTS. It is worth noting that AQL accepts queries in a free format. Thus, the format used in the examples is not mandatory; it has been adopted here only for the sake of readability. Finally, even though the use of APL has been kept to the minimum necessary to clarify the computations required in some examples, some knowledge of APL is needed; those not familiar with it can refer to [8].

Working with AQL

To introduce the structure and basic features of AQL, let us consider a simple query in which values are requested from a domain of a relation, while imposing a condition on the values of another domain. For instance,

"Find the budgets of the activities in department 145"

This query is formulated as

```
AQL 'PROJECTS'
(BUDGET OF ACTIVITY)
WHEN
DEPT EQ 145                    (1)
```

The function *AQL* is used to gain entry into the system; it accepts as an argument the name of the data base. This causes the terminal keyboard to unlock, waiting for input; input is terminated by an empty carriage return. In this case, the input is constituted of a structure called a "when-clause"; i.e., the query is composed of two distinct parts separated by the dyadic function *WHEN*. The left argument is the requested attribute (*BUDGET*) referred to the proper relation by the function *OF*; the right argument is an elementary condition made up of an attribute name (*DEPT*) followed by a comparison function (*EQ*) whose right argument is a constant. The comparison operation performed by the function *EQ* (and by its companions *GT*, *LT*, etc.) is extended to the case where the right argument is a list of values, which can also be the result of a previous or of a nested query. In this case the comparison operation is performed on the Cartesian product of the arguments, i.e., on the set of ordered couples formed by every value defined for the attribute left argument with each element of the right argument. This mode of operation is very useful when it is important to preserve a correspondence among values in the result and each value in the list. That is, for each element in the list right argument, a list of identifiers of the relevant tuples is determined. Each tuple identifier (tid) consists of an integer, which is biunivocally associated with each tuple of each relation. The *i*th element of the right argument is associated with the *i*th list of tuple identifiers, and vice versa. A parallel set of comparison functions (*GT*, *GE*, *EQ*, *NOTEQ*, *LE*, *LT*) provides for the "scalar" mode of comparison, i.e., every element of the left argument is compared with the corresponding element of the right one. Of course, both arguments have to be "conformable" in the APL sense. For this purpose, attributes are treated as three-dimensional arrays. The final result of the execution of the query is a three-dimensional APL array with, in this case, one plane and columns, which carries in each row the budget for the activities in department 145. Furthermore, a global variable, called *BUDGET*, is created in the workspace and contains the previous values. This side effect is produced only in the cases, like this one, in which the query is not nested in an outer one. The variable *BUDGET* is now available in the workspace for any further processing, as, for instance, to compute the total budget:

```
Z←+/[2]BUDGET
```

Of course, the primitive *APL* `'+/[2]'` can be replaced by a predefined user function *SUMUP*. Whenever all the attributes of a relation are in the request list, this can be expressed more concisely by the keyword *ALLINFO*; if we had requested *almost all* the attributes, the function *EXCEPT* would have been used in the following way:

```
(ALLINFO EXCEPT BUDGET) OF
                    ACTIVITY...
```

On the other hand, when no condition is imposed on the attributes (i.e., all tuples are wanted), then the function *ALL* replaces the condition list; it assumes, as its argument, the relation name in the same "when clause." The default option *WHEN ALL* is applied whenever only a request list is specified in the query. Formally, the functions *ALLINFO* and *ALL* correspond to Codd's "restriction" and "projection" operators.

An example of application of default options together with the use of APL primitives in the body of a query is

"Compute the total budget of all the activities"

```
AQL 'PROJECTS'
+/[2]BUDGET                    (2)
```

Since the attribute *BUDGET* (as it appears defined in the master relation) belongs only to the relation *ACTIVITY*, it is possible for the system to univocally qualify the attribute *BUDGET OF ACTIVITY*, and the default options transform the query into its equivalent and complete form:

```
+/[2](BUDGET OF ACTIVITY)
WHEN ALL ACTIVITY
```

The (only) final result of this query is a numeric value representing the sum of all budgets.

The left argument of the function *WHEN* can be a request list constituted of more than one attribute name catenated by the keyword *WITH*. Similarly, the right argument of *WHEN* can be a condition list consisting of a number of elementary conditions separated by the logical functions *AND*, *OR*, *NOT*. An example of this is the following request:

"Activity number and departments for those activities located in Bari and whose budget is between 1000 and 2000."

```
AQL 'PROJECTS'
((ACTNO WITH DEPT) OF ACTIVITY)
WHEN
(CITY EQ BARI) AND BUDGET
    BETWEEN 1000 WITH 2000          (3)
```

In this query, even though the attribute *CITY* belongs to the relations *ACTIVITY* and *SUPPLIER*, it is assigned on a simple preference basis to that relation which matches the one in the left argument of *WHEN* in the same when-clause. In this case the results of the query are twofold: first, a two-row character matrix containing the names of the requested attributes is produced as the explicit result; second, as a side effect, two variables with the names *ACTNO* and *DEPT* are initialized in the workspace.

To introduce some more primitives of the query language, and to show how they are integrated into the programming language, we now describe some problems of costs of activities and projects. Suppose we want to

"Compute the costs of those activities whose suppliers are located in Rome."

First we have to make a query asking for the unit price and quantity of the resources used in those particular activities:

```
AQL 'PROJECTS'
((UNITPRICE WITH QUANTITY)
OF RESOURCES)
WHEN
SUPPLIERNO ISONEOF
   (SUPPLIERNO OF SUPPLIER)
   WHEN
   CITY EQ ROME                     (4)
```

This example represents a typical way of navigating among relations and shows how nesting of queries is accomplished in *AQL*. The function *ISONEOF* performs a set membership operation. Its right-hand argument is a list of values (here given as the explicit result of the inner query), and the left-hand side argument is an attribute name, which, in this case, the system automatically refers to the relation *RESOURCES*.

Unlike the function *EQ*, the function *ISONEOF* does not preserve any correspondence between the elements of its arguments; i.e., a list right argument produces a one-row matrix of tid's, with no order correspondence left between the elements of the list and the tid's. Of course, if the right argument consists of one element only, *EQ* and *ISONEOF* behave the same. To compute the cost, we may use the two variables now available in the workspace:

```
COST←+/[2]UNITPRICE × QUANTITY
                                    (5)
```

Since this expression will become very common in any further computation of cost, it turns out to be useful to define a function for it. A convenient way for doing this is the following:

```
∇ R←ACTCOST M
[1]   R←+/[2](⍎M[1;]) × ⍎M[2;]
∇
```

The function *ACTCOST* is quite special: its argument (*M*) is a two-row character matrix; it performs the same computations as in (5) on the values obtained by executing the variable names contained in each row of the argument. Notice that the matrix *M* is exactly equivalent to the explicit result produced by the query (4), so it would be desirable to introduce this function directly in the query. In AQL this is possible, since the user can extend the set of basic primitives of the language simply by adding to them the new user-defined APL function by means of a system function called *ADDFNS*. In this way, not only the function name but also its syntactic nature is known by the system and by the AQL interpreter. In our case, this is achieved by typing

```
Z←ADDFNS'Z←ACTCOST M'
```

The result is either 0, if the operation was successful, or an error code. Once this has been done for the function *ACTCOST*, it can be used in a query, as in the following example:

"For each activity, list the activity number and its total cost."

```
AQL 'PROJECTS'
(DISTINCT ACTNO OF RESOURCES)
CAT
```

```
ACTCOST
((UNITPRICE WITH QUANTITY)
   OF RESOURCES)
GROUPBY ACTNO                    (6)
```

The function *CAT* is used to catenate the results of two queries at the same level of nesting. In this case its left argument extracts, from the relation *RESOURCES*, all the different values of the attribute *ACTNO*; this is a projection in Codd's sense [1]. The right argument is a grouping operation, a useful means for representing many-to-one relationships. Then the cost is computed after grouping, for each activity in the relation *RESOURCES*, the corresponding unit price and quantity referred to each resource used by that activity.

Some entities (relations) in the data base may have attributes (for instance, activity numbers in some departments or activities supplied with certain resources) from which it is possible to extract a subset of elements (e.g., departments) which correspond to *all* (or *only*) those instances of another attribute (e.g., project) which satisfy a given condition (e.g., city is Rome). The functions which allow the user to find those values are called quantifiers. This is a class of functions (namely, two: *HASALL* for *all*, *HASONLY* for *only*) implemented as primitives in AQL. An example of their use is given by the following problem:

"Compute the total cost of those activities developed in the departments which have *only* activities supplied by suppliers located in Rome."

In AQL this request may be formulated as

```
      AQL 'PROJECTS'
ACTCOST
((UNITPRICE WITH QUANTITY)
   OF RESOURCES)
WHEN
DEPT HASONLY ACTNO EQ
   (ACTNO OF RESOURCES)
   WHEN
   CITY EQ ROME                  (7)
```

With this formulation of the query, the semantic completion routine detects that the attribute *DEPT* can be referred only to the relation *ACTIVITY*, which is different from the request relation in the same when-clause, i.e., *RESOURCES*. In this case at least one intermediate query should be built up in order to "link" the two relations. The system can carry out this process by entering the so-called inference mode. In order to activate this process, an AQL function has been defined by which it is possible to describe the correlations that in a given data base exist among relations and that may be represented as a graph in which the nodes correspond to the relations and the arcs to the link attributes. Figure 3 shows such a graph for the data base *PROJECTS*. This graph is represented in the system by means of a navigation matrix, which carries, for each ordered couple of relations, the couple(s) of linking attributes. These links should be explicitly established by the data base administrator, since choices based on world knowledge and data base semantics are involved. In general, a link involves a couple of attributes which are semantically equivalent, in the sense that they represent instances of the same set of entities, which is their underlying domain. This implies that, in general, there may be more than one couple of link attributes between two relations; in this case all the inference paths are present in the navigation matrix, and the disambiguation choice is asked of the user.

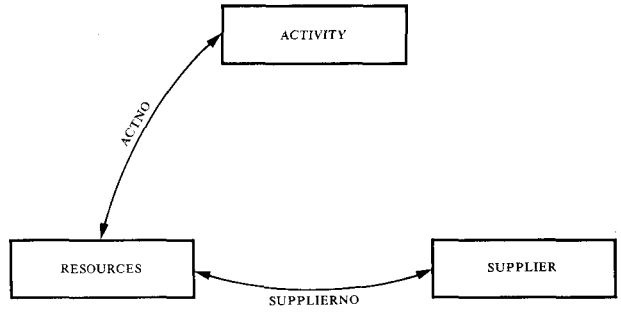

Figure 3 Correlation among relations.

In the example on hand a one step inference path is sufficient to connect the two relations *RESOURCES* and *ACTIVITY* by means of the link attribute *ACTNO*. The query is now restated by the system as follows:

```
ACTCOST
((UNITPRICE WITH QUANTITY)
   OF RESOURCES)
{WHEN
 ACTNO ISONEOF
   (ACTNO OF  ACTIVITY)}
```

```
WHEN
DEPT HASONLY ACTNO EQ
  (ACTNO OF RESOURCES)
WHEN
CITY EQ ROME                    (8)
```

Here the braces denote the part of the query built by the inference mechanism. The query is not yet ready to be executed since the attribute *CITY*, left argument of the function *EQ* in the rightmost condition, cannot be referred to the request relation *RESOURCES* in the same (innermost) when-clause. Moreover, it belongs to both the relations *SUPPLIER* and *ACTIVITY* so its qualification cannot be decided by the system. To cope with this kind of ambiguity, the user is prompted with a menu asking him to choose the proper relation:

```
...(ACTNO OF RESOURCES)
   WHEN (CITY OF...) EQ ROME
CHOOSE THE NUMBER
CORRESPONDING TO THE PROPER
RELATION
1.SUPPLIER
2.ACTIVITY
□:
1    (user's answer)            (9)
```

The relation *SUPPLIER* (to which *CITY* can now be referred) is different from the request relation; again the system enters the inference mode, and, accessing the navigation matrix, tries to build the intermediate query linking the two relations. The final result of this process transforms the query into

```
ACTCOST
((UNITPRICE WITH  QUANTITY)
  OF RESOURCES)
{WHEN
 ACTNO ISONEOF
   (ACTNO OF ACTIVITY)}
WHEN
DEPT HASONLY ACTNO EQ
  (ACTNO OF RESOURCES)
{WHEN
 SUPPLIERNO ISONEOF
   (SUPPLIERNO OF SUPPLIER)}
WHEN
CITY EQ ROME                    (10)
```

The expression is equivalent to the internal representation of the query which can now be executed.

The structure of the projects, as it is defined in Fig. 2, suggests that the cost of an activity is different from the cost of the project to which that activity belongs, except for those projects with no subsidiary activities. Thus a problem such as

"Compute the cost of a given project"

is more general than the previous ones and implies for the user the definition of some "application" functions. The first step in solving this problem is

"Find all activities which depend upon a given project."

To find the activities which directly depend upon a given project, say *P*, simply means, in AQL, writing a query like

```
(ACTNO OF ACTIVITY) WHEN
                PROJECT EQ P    (11)
```

If we were interested in a list of projects (irrespective of their order), we would write *ISONEOF* instead of *EQ*. This query should then be applied again, descending the tree or sub-tree whose root is *P*, and saving the result at each step. To avoid recalling the AQL interpreter at each step of this recursive process, we can *name* the query. The facility of naming a query is equivalent to that of naming an expression in APL, i.e., defining a function. This function is defined as the APL expression equivalent to the result produced by the AQL interpreter; moreover, the system automatically adds its name to the basic primitive set of AQL. Once a name has been assigned to a query, it can be saved for later use, it can be executed by referencing its name, or it can even be introduced in another query as a primitive of the language.

This facility can be used for naming the query (11), as follows:

```
'Z←SONS P' AQLDEF 'PROJECTS'
```

```
(ACTNO OF ACTIVITY)

WHEN

PROJECT EQ P
```

The left argument of the function *AQLDEF* is the header of the function to be defined, and its name (*SONS*) is the *name* of the query whose execution can be performed (say for P equal to 239) simply by writing the expression *SONS* 239. The descendent(s) of the tree or sub-tree whose root node is a given project P is (are) retrieved by the following recursive function:

```
∇ Z←DEPENDENT P

[1] Z←P                    Set up the result.
[2] →0 IF EMPTY P←SONS P
                           if P has no sons, then quit;
[3] Z←Z ON DEPENDENT P
                           else compute its dependents,
∇                          and stack on the previous
                                 ones.
```

(12)

The result of this function is the list of all activities dependent upon P. The cost of a list of activities can be computed by defining the function:

```
'Z←COST P' AQLDEF 'PROJECTS'

ACTCOST

((UNITPRICE WITH  QUANTITY)

  OF RESOURCES)

WHEN

ACTNO EQ P
```

(13)

The cost of a project is then given by the expression

```
COST DEPENDENT P
```

(14)

The function *COST* can be redefined whenever the problem is formulated in a different way; for instance, if *COST* is the name of the query, it would be defined as

```
ACTCOST

((UNITPRICE WITH  QUANTITY)

  OF RESOURCES)

WHEN

(ACTNO EQ P) AND CITY EQ ROME
```

The expression (14) gives the cost of the project P only for those activities located in Rome or whose suppliers are located in Rome, since *CITY* belongs to both the relations *SUPPLIER* and *ACTIVITY*. This ambiguity is resolved during the execution of *AQLDEF* by use of the menu, and then the query completed by the inference process.

In the previous examples there is no way of knowing whether a given project is the root of the tree (i.e., the ancestor of all its activity) or whether it is a son of another project. This problem may be stated as follows:

"Compute the total cost of the 'root' project to which a given project or activity belongs."

In this case, starting from an activity we have to move upward on the tree to which it belongs, to the root. First of all, using the naming facility, it is easy to define a function for finding the "father" (direct ancestor) of an activity:

```
'Z←FATHER A' AQLDEF 'PROJECTS'

(PROJECT OF ACTIVITY)

WHEN

ACTNO EQ A
```

(15)

Then a recursive function for re-ascending the tree up to the root can be defined as

```
∇ Z←ROOT A;FA

[1] Z←A                    Set up the result
[2] →0 IF EMPTY FA←FATHER A
                           if A has no father then quit,
                           that's the root;
[3] Z←ROOT FA              else go up another step.
∇
```

(16)

The problem is solved by the expression

```
COST DEPENDENT ROOT A
```

(17)

This means, for any given activity A, find the root project to which A belongs, explode this project in all its depending activities, and finally compute the cost. This algorithm exemplifies a broader class of algorithms, which are appli-

cable whenever the elements of a couple of attributes of the same relation (in our case, *PROJECT* and *ACTNO*) are drawn from the same domain. They then form a relation in an algebraic sense (see, for example, [11]), and such operations as product, power, and transitive closure (illustrated above) may be defined.

With a simple generalization of the functions *DEPENDENT* and *ROOT*, it is possible to generalize the last problem as

"Compute the total cost for all the root projects,"

which is solved by the expression

```
COST DEPENDENT ROOT DISTINCT
        PROJECT OF ACTIVITY          (18)
```

which, first of all, finds all the different projects (*DISTINCT PROJECT OF ACTIVITY*), then finds the respective roots (*ROOT*), retrieves their "sons' " activities (*DEPENDENT*), and then computes their cost (*COST*).

This is a style the user can adopt in solving problems or building his own applications with AQL, even though it is not the only one. Sometimes the user may find it convenient or preferable not to name all queries involved in his application functions, for instance, when the query is to be executed only once in the application, or when the application is to be run only periodically (i.e., weekly or monthly), as in the case of producing inventory or cost reports. In this case the query can be directly introduced in the body of the application function by the dyadic function *AQLS*, whose left argument is the data base name and whose right argument is a character string representing the query. Since *AQLS* is the dyadic equivalent of the function *AQL*, the query will be interpreted and executed inside the calling function.

Let us consider the problem of producing a cost report such as

"For a given project find all its suppliers, and for each supplier compute the total cost of the resources supplied."

An application function which solves this problem is here described in a step-by-step fashion. Define a monadic function, with some local variables:

```
V T←COSTREPORT P;AP;SP;C
```

Find the activities depending upon *P*

```
[1]  AP←DEPENDENT P
```

Find the supplier of the activities *AP*. Since an activity can be supplied with different resources by the same supplier, the function *DISTINCT* suppresses duplicates of suppliers' numbers for each activity:

```
[2]  'PROJECTS' AQLS
     'SP←DISTINCT
     (SUPPLIERNO OF RESOURCES)
     WHEN ACTNO ISONEOF AP'
```

In *SP*, we obtain the list of the codes of suppliers who supply *at least* the activities *AP*. Now, we have to find the cost for those suppliers and those activities:

```
[3]  'PROJECTS' AQLS 'C←ACTCOST
       ((UNITPRICE WITH QUANTITY)
         OF RESOURCES) WHEN
       (SUPPLIERNO EQ SP)
       AND ACTNO EQ AP'
```

Finally, put the results in a tabular form:

```
[4]  T←P CAT SP CAT C
     ∇
```

To produce the same kind of report for the project ancestor of *P*, the following expression should be executed:

```
COSTREPORT ROOT P                    (19)
```

To produce a cost report for all the "root" projects in the relation *ACTIVITY*, the expression is

```
COSTREPORT ROOT DISTINCT
        PROJECT OF ACTIVITY          (20)
```

with obvious meaning of the functions.

The facilities of AQL that we have described so far turn out to be very useful tools for the user, not only in writing his own application functions, but also in some cases where the formalization of (very) complex (or ambiguously formulated) queries can be found more easily by breaking down the original request into a series of simpler ones. This technique not only helps in finding the solution but also gives the user a useful means for gaining insight into the nature of his problem, as well as verifying some properties of his data. As an example of this, consider the following query:

"Find those activities such that the city where they are taking place is the same as the city of all their suppliers."

A possible way in which this request can be worked out is given by the following sequence of steps:

1. Find all distinct activities present in the table *RESOURCES*:

```
DA←DISTINCT
    ACTNO OF RESOURCES                    (21)
```

2. Group the suppliers (in *RESOURCES*) by the activities they supply, i.e., obtain an array carrying for each activity the list of its suppliers:

```
SBYA←(SUPPLIERNO
   OF RESOURCES)
GROUPBY
ACTNO                                     (22)
```

3. Find from the table *ACTIVITY* the cities where the activities *DA* are developed:

```
CA←(CITY OF ACTIVITY) WHEN
        ACTNO ISONEOF DA                  (23)
```

4. Find the same for each supplier in *SBYA*:

```
CS←(CITY OF SUPPLIER) WHEN
SUPPLIERNO ISONEOF SBYA                   (24)
```

5. The two variables *CA* and *CS* are three-dimensional arrays; furthermore, *CA* is a one-plane array (as activities take place in one city at most), with the number of rows equal to the number of planes in *CS*. These two conditions hold only if there is one city for each supplier and activity, and each activity has at least one supplier. Note that this is a way to do some data analysis or to verify the consistency and/or integrity of the information in the data base. In this step, now, we have to select from *DA* those activities such that the corresponding plane in *CS* has all the rows equal and the value is equal to the corresponding element of *CA* (i.e., the suppliers of that activity live in the same city as the activity). This would be done in the following way by a more formal APL style: Given the set of acvities *DA* and an integer index *I* such that *I*∈⍳(⍴*DA*)[2] (ranging from 1 to the number of rows in *DA*), the activity *DA*[;*I*;] should be selected if the expression

```
∧/CA[;I;]∧.=CS[I;;]
```

holds; i.e., the name of its city is found in all the rows of the plane carrying the names of the cities of all its suppliers. By using basically this expression it is possible to define a function for comparison between *CA* and *CS*, with the syntax

```
V←CA EQALL CS                             (25)
```

The result *V* is a logical vector with as many elements as the rows in *CA* and *DA*.

6. Finally, the logical vector *V* can be used to select the array of activities *DA* as:

```
V/[2]DA
```

whose result gives the desired activities whose city is the same as that of all corresponding suppliers.

The characteristics of the host language give to the user the possibility of executing the above steps directly at the terminal in an interactive way and at each step deciding what to do next. Using the open-endedness facility, the function *EQALL* can be added to the basic set of the language, and the request can be treated as a single query which can also be named for further use. The formulation of the query is obtained by putting together the previous steps:

```
        AQL 'PROJECTS'
(((CITY OF ACTIVITY)
WHEN
ACTNO ISONEOF DA)
EQALL
(CITY OF SUPPLIER)
WHEN
SUPPLIERNO ISONEOF
   (SUPPLIERNO OF RESOURCES)
   GROUPBY ACTNO)/[2]
     DA←DISTINCT
   ACTNO OF RESOURCES                     (26)
```

It is worth noting here the use, in the body of the same complex query, of a variable (*DA*) specified as the result of an operation (*DISTINCT*) and referenced as an argument of another one (*ISONEOF*). This is a quite common way in AQL to avoid, whenever possible, duplication of operations (speeding up the interpretation and execution of the query), since a variable name can always be used inside a query.

In AQL it is possible to reference the names of attributes by their synonyms; this facility is a particular case of a more general mechanism which allows the user to define new attributes as functions of pre-existing ones. The use of these so-called defined attributes is particularly helpful whenever some combinations of attributes in a given relation define concepts that the user may wish to reference as if they were primitive ones in the formulation of queries. The definitions are always expressed as APL functions; the procedure to introduce definitions in the system, and the protocol (i.e., the conventional way of writing the definitions to interface with the interpreter) are illustrated in the next section. By using the definition mechanism, it is possible to define in the relation *ACTIVITY* the attribute "starting date" (*SDATE*) as a function of *YEAR*, *MONTH*, and *DAY*, and the attribute "age of an activity" (*AGE*) as a function of its starting date *SDATE*. The use of defined attributes in a query is shown in the following example:

"Which activities will be at least 3 years old at the end of 1978."

```
      AQL 'PROJECTS'
(ACTNO OF ACTIVITY)
WHEN
AGE GE 3 AT 1978 12 31                    (27)
```

The dyadic function *AT*, which accesses the time stamp, thus getting the date (year, month, and day) when the query is executed, causes the left argument to always conform to a three-element vector which is assumed to be the number of years, months, and days (this is the conventional way of expressing dates in AQL); takes as a right argument a date; and returns as a result the age computed at the date given by the time stamp. The query is restated by the system, applying the definition of *AGE* as:

```
(ACTNO OF ACTIVITY)
WHEN
((YEAR LT Z[;1]) OR ((YEAR EQ
    Z[;1]) AND ((MONTH LT Z[;2])
OR ((MONTH EQ Z[;2]) AND ((DAY
                    LT Z[;3]) OR
(DAY EQ (Z←TIMEAGO 3
AT 1978 12 31) [;3])))))          (28)
```

which merely reads:

"Activities which started less than three years before 1978, or which started exactly three years before 1978 but in a month preceding December, or, even though they started in December 1975, at least they started some days before December 31th, or, finally, those started exactly on December 31th, 1975."

The function *TIMEAGO* computes, from the date indicated by the time stamp, the age referred to the data given as its right argument. The result is assigned to *Z* as a three-column matrix of number of years, months, and days. This reformulation of the query shows how the definition of *AGE* depends on the comparison function, of which it is the left argument; in fact, *GE* has been transformed into *LT*. The definition of an attribute can be, in general, different if it is referenced at the left or the right argument of the function *WHEN*. For instance, again using the domain *AGE*, one can ask for

"Age of activities located in Bari."

```
      AQL 'PROJECTS'
(AGE OF ACTIVITY)
WHEN
CITY EQ BARI                              (29)
```

Again, reformulation of the query shows the definition of *AGE* applied in this case to be:

```
TIMESINCE
((YEAR WITH MONTH WITH DAY)
  OF ACTIVITY)
    WHEN
    CITY EQ BARI                          (30)
```

The function *TIMESINCE* computes the time since a given date up to the date in the time stamp; the result is a matrix, containing, in each row, the years, months, and days. The result of the query is an APL variable called *AGE*, corresponding to the requested domain; this means, in other words, that externally (i.e., from a user point of view) the defined attributes always behave as the basic ones. A typical application in which defined attributes are specially useful for the data base we are considering is the following:

"How long will a given project *P* last?"

This problem implies that, to find the residual duration of a project *P*, we have to descend the tree (or the sub-tree)

whose root is P and, at each step, compute the residual duration of its depending activities. The nature of the problem suggests, again, the use of a recursive function.

Suppose we call $RESDUR(P)$ the residual duration of P; then the essential steps of the procedure to compute $RESDUR(P)$ can be informally described as

1. If P is empty, then $RESDUR(P)$ is 0, and we can quit.
2. Else we have to check whether P is started; if this is the case, then $RESDUR(P)$ is defined as

```
(SDATE(P) + DURATION(P))
                    - TODAYDATE;
```

3. Else (i.e., P is not yet started) $RESDUR(P)$ is defined by

```
DURATION(P) + MAXIMUM
                    RESDUR (SONS(P)).
```

Here $SDATE(P)$ and $DURATION(P)$, which represent the starting date and the duration of the project P, should be computed by issuing a query against the relation $ACTIVITY$. Since this query will be introduced in the application function and executed at each step of the recursion, it can be named

```
'Z←DATEDUR P' AQLDEF 'PROJECTS'
((SDATE WITH DURATION)
  OF ACTIVITY)
WHEN
ACTNO EQ P                    (31)
```

The expression $SONS(P)$ represents the name of the query (already defined) which finds the activities directly depending upon a given project P, and finally $TODAYDATE$ is an APL nulladic function that returns the value of the time stamp.

The three steps used to describe the problem informally can now be the guideline to writing the application function $RESDUR$ whose definition, together with some comment lines, is shown in Appendix A.

With the same style used in previous examples, the function $RESDUR$ can be used, for example, to compute

"The residual duration of the 'root' project to which activity A belongs."

In fact, the expression

```
RESDUR ROOT A                 (32)
```

solves the problem.

Data definition

Data definition facilities have been introduced in AQL as a set of functions, with the same syntax as the query functions, to allow the data base administrator to define or drop attributes of a relation, to add tuples (rows) to a predefined relation, as well as to create or drop copies and views from basic relations. In this section we describe the syntax and properties of the functions for data definition through a series of examples from the data base in Fig. 1. The first operation to create a new relation is that of defining its schema (e.g., attribute names, definitions, and types). This operation is performed by the function $DEFINE$; an example of its use is the following:

```
AQL 'PROJECTS'
DEFINE (ACTNO WITH PROJECT WITH
  CITY WITH DEPT) OF ACTIVITY   (33)
```

Here we ask that the relation whose name is the right argument of the function OF be associated with a list of attribute names in its left argument. The function $DEFINE$ interactively asks the user to specify, for each attribute, the type and the definition (if one exists). The attribute type value is mainly used to establish the ordering relationship when an index may possibly be built for that attribute. The ultimate and most important effect of the execution of $DEFINE$ is that the master relation is updated with the description of this new relation. To show how the mechanism of defined attributes works, suppose we want to

"Add to the pre-existing attributes of the relation $ACTIVITY$ a new one called $ACTCODE$, as a synonym of $ACTNO$."

This operation is performed by the expression

```
AQL 'PROJECTS'
DEFINE ACTCODE OF ACTIVITY    (34)
```

The dialog between the system and the user is in this case

```
S (system)  ATTRIBUTE-NAME IS ACTCODE
S           TYPE:
U (user)    N
S           DEFINITION:
U           MSYNS 'ACTNO'
```

where $SYNS$ is an APL function which declares $ACTCODE$ to be a synonym of $ACTNO$; it represents,

also, a typical example of the protocol for writing definitions.

The major idea underlying the mechanism of defined attributes is the following: Definitions are APL functions, for which each execution always produces an AQL statement. This means that, in order to operate properly, this function should know the syntactic context in which the manipulation of the query should be performed to account for the definition. The information that defines the syntactic context is simply the function name, of which the defined attribute is an argument, together with its arguments in the parse tree. This context is what we have called the protocol or the interface between the definition and the AQL interpreter, and it is always specified as the left-hand side argument of the definition function. This is the reason why all these functions should be dyadic, with the right argument referred to the defining attribute.

It is the task of the AQL interpreter, before executing the definition, to associate with the left argument the value of the syntactic context, by representing it as a three-row matrix containing, respectively, the left argument (possibly empty), the function name, and the right argument. The AQL statement produced as the result of executing the definition is parsed and checked for semantic completeness by the interpreter before it is substituted into the old syntactic context. A very simple way to define the function *SYNS* is

```
     ∇ Z←A SYNS TRUEDOM
[1]    Z←TRUEDOM,A[2;],A[3;]
     ∇
```

Then, knowing only the protocol, the user can write the definition of an attribute as an APL function and add to the master relation the newly defined attribute using the function *DEFINE*.

AQL provides a function to store the data into a predefined relation; this operation is performed by the function *ADD*, whose syntax is shown in the next expression for the case of adding new tuples to all the domains of the relation *ACTIVITY*.

```
AQL 'PROJECTS'
(ALLINFO OF ACTIVITY)
ADD VALUES                                    (35)
```

The left argument always specifies the attribute and relation names; the variable called *VALUES*, in the right argument, can be:

1. The name of a preformatted file, in which case the *ADD* function performs a bulk input kind of operation.
2. A character matrix, representing a list of names of global variables which contain the attributes' values.
3. A three-dimensional array, in which each plane carries the values of the attributes whose names are in the list left argument.
4. An empty APL vector; then the function will accept the values of each attribute of the relation in an interactive way.

Beforehand, this function prompts the user with a list of options asking him to choose for which domains an inversion should be built.

For the management of the inversions the data base administrator can have at his disposal the two functions *BUILDINDEX* and *ERASEINDEX* for, respectively, building or dropping the inversion of a given domain. Both functions accept, as arguments, the attribute and relation names in the usual way:

```
AQL 'PROJECTS'
BUILDINDEX ACTNO OF ACTIVITY                  (36)
```

The problem of deleting attributes from a relation, or even the relation itself, is dealt with in AQL by the monadic function *DROP*. The expression

```
AQL 'PROJECTS'
DROP (UNITPRICE WITH QUANTITY)
     OF RESOURCES                             (37)
```

physically deletes the attributes and their descriptions from the master relation; all the attributes which have been defined as functions of at least one of the two dropped attributes will be flagged as "no longer usable." To cancel a relation completely, one can write

```
AQL 'PROJECT'
DROP ALLINFO OF ACTIVITY                      (38)
```

There are two other important facilities of data definition which offer to the user the possibility of extracting from the set of relations which constitutes the data base those particular subsets he will use more frequently to perform tests and simulation and which he can consider as "private" relations. Two classical ways are available to meet these needs: the use of copies and/or views extracted from basic relations [12].

Both copies and views are supported by AQL, and in the following text we give some examples of both. Suppose the user wants to

"Create a copy, from the relation *ACTIVITY*, consisting of activity numbers, projects, and starting dates of all the activities located in Rome and call this copy *ROMEACTIVITY*."

This is performed by

```
AQL 'PROJECTS'
((ACTIVITYNO WITH PROJECT WITH
        DATE) OF ROMEACTIVITY)
ISCOPYOF
((ACTNO WITH PROJECT WITH SDATE)
                OF ACTIVITY)
WHEN
CITY EQ ROME                    (39)
```

The right argument of the function *ISCOPYOF* can be either a character matrix or a three-dimensional array, with the same meaning as for the function *ADD*. Note that both cases cover the possible result of a query, which can be used as the right argument of *ISCOPYOF*. Of course, this function can be thought of as a combination of the two functions *DEFINE* and *ADD*; in fact, it physically builds a new relation for the user and marks it as "private." An important aspect of this example is the fact that, while *SDATE* in *ACTIVITY* is a defined attribute, in the new relation *ROMEACTIVITY*, the corresponding attribute *DATE* has an empty definition. This is due to the fact that *DATE*, being constituted of the result of a query on the attribute *SDATE*, has each item directly obtained from the catenation of the values in *YEAR*, *MONTH*, and *DAY*. The mechanism of building copies can be a useful means for approaching the problem of having different levels of authorizations in the data base. For instance, a user who is not authorized to update the basic relations, but only to define copies of them, being the owner of the copies, automatically has the maximum level of authorization, i.e., he is the administrator of his personal data base. Furthermore, he can also authorize other users to read and/or update, etc., his relations. It is worth noting that copies are a static way of looking at (parts of) basic relations, since any change of these does not affect the copies and vice versa. The way of coping with this problem is to introduce the concept of views. Views give the user the possibility of looking at the desired subsets of the basic relations without physically building new ones. Unlike what happens with copies, views are dynamically sensitive to the updates made in their underlying relations.

The way to define a view, in AQL, is very similar to that of defining copies. Suppose we want to

"Look only at those activities, projects, and budgets which are in department 130."

Then the view is built by the expression:

```
AQL 'PROJECTS'
((ACTIVITY WITH PROJECT WITH
        BUDGET) OF DEPT130)
ISVIEWOF
((ACTNO WITH PROJECT WITH
        BUDGET) OF ACTIVITY)
WHEN
DEPT EQ 130                     (40)
```

Here the left argument of the function *ISVIEWOF* describes, as usual, the names of the attributes and their relation, to be defined as a view, whereas the right argument is the query which gives the view definition. The basic idea used in AQL for dealing with views is that of treating each attribute in the view as if it were a defined attribute. This definition is obtained from the result of interpreting the query which defines the view and extracting that portion of the parse tree which refers to the corresponding requested attribute(s) in the underlying relation. This correspondence is established on the basis of the order in which the attribute names appear both in the view and in the list of requested attributes in the query. With this approach the execution of the query defining the view will never take place, i.e., the values of the requested attributes will not be retrieved from the relation; the view definition will be used to modify, in a suitable way, the parsed form of the queries made against the views. It is also possible in AQL to define views derived from more than one relation; for instance, the previous view can be redefined by adding the cost of the activities.

```
AQL 'PROJECTS'
((ACTIVITY WITH PROJECT WITH
    BUDGET WITH COST) OF DEPT130)
ISVIEWOF
(((ACTNO WITH PROJECT WITH
        BUDGET) OF ACTIVITY)
WHEN
```

```
DEPT EQ 130)

CAT

ACTCOST ((UNITPRICE WITH

        QUANTITY) OF RESOURCES)

WHEN

ACTNO ISONEOF

      (ACTNO OF ACTIVITY)

      WHEN

      DEPT EQ 130                    (41)
```

Note that, since the query for computing a cost of a given activity has been named in the previous section (13), it can be used directly as the left argument of the function *CAT* in the same way as in (6). Then we can write the expression

```
COST (ACTNO OF ACTIVITY) WHEN

                DEPT EQ 130            (42)
```

as the right argument of the function *CAT*.

Views, as well as copies, behave from an external view point as basic relations on which the user can apply all the query language facilities described in the previous section. For instance, we can ask to

"Find those activities in department 130 whose cost is greater than the respective budget."

```
      AQL 'PROJECTS'

(ACTIVITY OF DEPT130)

WHEN

COST GT BUDGET                         (43)
```

Note here the use of the scalar comparison function, necessary because each value of the cost should be compared only with the corresponding value of budget.

As far as update through the view is concerned, to avoid the problems exposed in [13], we choose to allow it only when the view is obtained from only one relation and contains its master key.

Update facilities

Of the three update operations, i.e., insert, replace, and delete, the first one is performed by the function *ADD*, described in the previous section, whereas the last two are performed by a function called *REPLACE*. This is a dyadic function whose right argument should be a query in which attribute names to be updated are specified, possibly together with conditions specifying which particular values should be affected by the operation. The left argument is an APL array resulting from a function or an expression having the same structure as the explicit result of a query, and therefore it can also be a query. It carries the values that will be replaced in the positions specified by the right argument. When a deletion should be performed, then the left argument is an empty APL object. To show how the update operation works in AQL, suppose that we want to

"Increase by 10 percent the budget of those activities with the maximum duration."

```
      AQL 'PROJECTS'

(1.1 × BUDGET) REPLACE

            (BUDGET OF ACTIVITY)

      WHEN

      DURATION EQ MAX DURATION

                                       (44)
```

In this case, the result of the query execution, which is the variable *BUDGET*, is directly used in the left-hand side argument of the function *REPLACE*.

An example of a deletion operation is

"Delete all the activities which have not yet started and with duration greater than 900 days."

```
     AQL 'PROJECTS'

'' REPLACE (ALLINFO OF ACTIVITY)

           WHEN

           (SDATE EQ 0) AND

           DURATION GT 900             (45)
```

which corresponds to deleting the tuples from *ACTIVITY* which satisfy the condition.

During the execution of the function *REPLACE*, a set of options is available to the user to allow him to control the operation in a stepwise fashion or to verify the data he is updating. To show how this mechanism works, we consider the following example:

"Update the values of the attributes year, month, day, and duration for project 20."

```
      AQL 'PROJECTS'

LISTVARS REPLACE
```

```
((YEAR WITH MONTH WITH DAY WITH
                    DURATION) OF ACTIVITY)
WHEN PROJECT EQ 20                    (46)
```

Here *LISTVARS* is an APL matrix containing a list of names of predefined variables with the new values. Furthermore, the function *REPLACE* requires that the number of these variables be equal to the number of requested attributes in the query. The list of options which are shown to the user before the replace operation actually takes place is

(1) *DISPLAY* variable-name

(2) attribute-name *STEP n*

(3) *ALL* attribute-name

With the first option, the user can display either values retrieved for a given attribute (in this case the argument of the function *DISPLAY* is a name of an attribute) or the values of any of the variables in *LISTVARS*. Thus, for example, if he wants to see the values of *YEAR*, he enters

```
DISPLAY YEAR
```

The second option gives to the user more direct control of the execution of the operation; in fact he can request that the update for the attribute specified in the left argument of the function *STEP* take place with *n* items at a time. Referring to the above example, we can choose to update the attribute *DURATION*, one item at a time, in which case the option is expressed as

```
DURATION STEP 1
```

The system prints one item of the attribute and accepts from the user the new value. Any time the user enters an empty value, the item is deleted.

Whenever the third option is specified, the update for the attribute specified as an argument of the function *ALL* takes place without further user control. *ALL*, for all the specified attributes, is the default option.

Conclusions

We have described how queries, data definitions, and updates are handled in AQL with a simple and unified syntax, which is the same as that of its host language, APL. The examples and problems given show that AQL can be used either by the nonspecialist or by the sophisticated APL user. It offers to both classes of users, together with a complete query facility, a very powerful means for problem solving and application building, allowing the user to always work in a homogenous environment through a complete integration with the host language. Different styles for building application functions have also been described to show how functions of different applications can be used, always applying the same syntax, to build new and more complex applications or to generalize existing ones.

AQL is undergoing further development to introduce new facilities which have not been described here, namely, functions for data control. These will cope with the problems of data security and authorization, as well as assertions about properties of data belonging to a given attribute for the purpose of controlling data consistency.

Appendix A: System architecture

In this appendix we give a functional description of the overall system, concentrating on *what* the various functions and their interrelationships are, rather than on *how* they are actually implemented in the prototype. From an architectural viewpoint, the system has a "layered" or multilevel structure constituted of three levels or schemata as in Fig. 4. Each level is characterized by a language for communicating, a set of objects corresponding to particular data structures or models on which it operates, and a set of functions which constitute its processing capabilities and environment.

The external level realizes what is known in the data base literature [14, 15] as the "conceptual schema," i.e., it is the level which supports the user's logical view of data and provides data independence capabilities. At this level the objects are simply sets of tables defined by their names and by their corresponding attributes. A table can be either a basic relation or a "user view," defined by manipulating one or more underlying basic relations [this mechanism was shown in the data definition section; see expressions (40) and (41)]. At this level communication with the user takes place through the query language.

The functions performed at this level are mainly those of interpreting and checking the query; in doing this job communication is established with the internal level by mapping tables and attribute names into their corresponding objects at the lowest level. The interpreter looks at the query as a character string, which is parsed and then checked for syntactic correctness and semantic completeness. The parser produces a complete parse tree of the query, using APL-like criteria: first, the right-most, least-nested function is dealt with, then the same procedure is recursively applied to the right and, possibly, to the left argument of this function. In this process syntax errors are detected and displayed to the user. The parsed form of the query is then checked for semantic completeness; in this phase, for instance, default options are applied and attributes are qualified by referring them to the proper relation. As in other systems (e.g., Codd's GAMMA-0 [16]), there is in AQL a master relation, i.e., a kind of catalog, which contains information describing all the relations be-

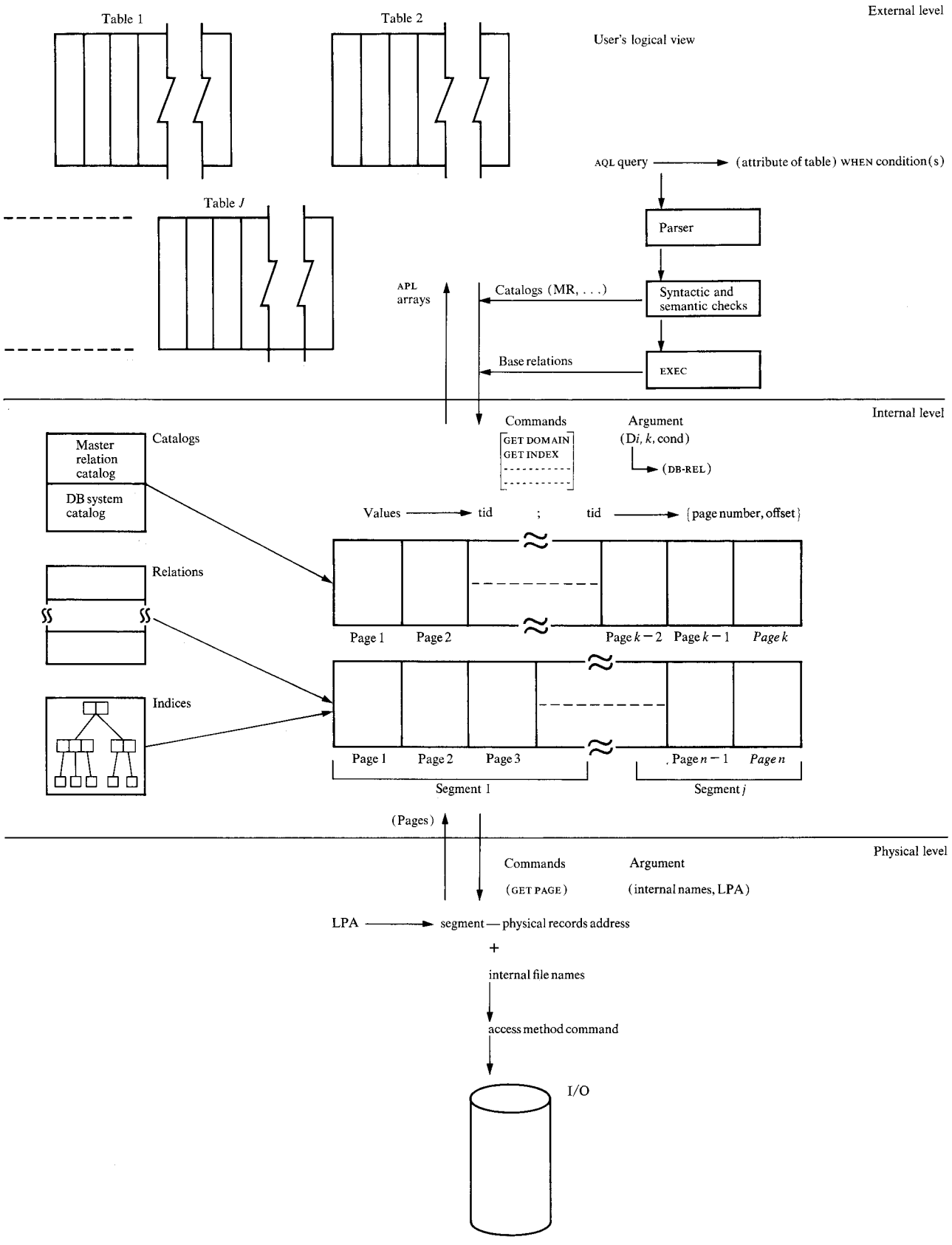

Figure 4 Overview of system.

longing to a data base, and it is accessed in this step. The final result of the interpretation is a canonical representation of the query, which can be fixed in the workspace and executed as an APL function.

Communication with the internal level takes place through a series of command functions. Each command can have one or more arguments. At this level the arguments are: data base name, relation name, attribute(s) name, and a set of values to look for. The internal level guarantees a very "loose" coupling between the user at the external level and the physical level of data representation; in fact, it provides an internal model of memory space which is an abstraction of the physical address space on which data are stored; in this way, it improves the degree of data independence.

From the viewpoint of the internal model, memory is organized as a set of pages which constitutes a linear address space. Pages correspond, at the physical level, to aggregations of records, and each page is saved on disk as a contiguous sequence of records of equal size. This technique has the advantage of simplifying buffer management at the physical level. A memory space that spans a continuous address space containing one or more pages (i.e., a record aggregate) is called a segment. A data set is simply a collection of segments, while a single collection of data sets described by an internal schema (which is part of the system catalog) defines a data base.

In this model there is a one-to-one correspondence between relations and segments, established at data base generation time. At this level a relation is seen as a set of domains (where each domain contains all the occurrences of values of the corresponding attribute), together with their corresponding indices or inversions, which represent a correspondence between domain values and the identifier of the tuple they belong to. Relations are stored as transposed files, i.e., by clustering the elements of a given attribute belonging to different tuples, instead of clustering the elements of different attributes belonging to the same tuple. This means that each page contains only elements which are homogeneous, since they are all occurrences of values of the same attribute. This technique has the advantage of speeding up retrieval when the number of relevant occurrences for a given query is greater than the number of attributes in the target list. The data structure used for the indices is that of a B-tree (as defined in [17]) having variable-length keys.

During the creation of a relation, the system builds for each domain the so-called tuple identifier to logical address converter (tilac); this is a mapping mechanism between intervals of tuple identifier (tid) values and logical page addresses. Other important objects at this level are the two catalogs. These are defined as special relations which describe the characteristics of all the objects of the internal schema. The master relation catalog describes all the relations belonging to any given data base; it holds information such as relation names, attribute names, types, and definitions (this last is a facility which gives the user the possibility of defining new attributes as functions of already existing ones; e.g., synonyms are a special case of definition). The data base system catalog is used to describe each data base in terms of the relations belonging to it and of the identifier of each relation creator; it also holds other system dependent information.

The functions performed at the internal level are mainly those of computing the logical paths (up to the value of page addresses) in order to access a set of data items both for querying and updating. The command functions available to manipulate data items are of different types, but the most significant ones are those to access the inversion and the domain of a given attribute. Concerning the index, we may (informally) write:

GET INDEX (D_j, K, cond) for read

GET INDEX (D_j, K, cond) for write

This function, using the B-tree of the index of domain D_j, finds the set of tuple identifiers (tid's) of D_j corresponding to the value K and for a given condition (e.g., equal, greater than, etc.). The command functions

GET DOMAIN (D_i T) for read

GET DOMAIN (D_i, T) for write

extract from the domain D_i the set of values corresponding to the tuple identifiers given in T. The result returned is an APL array with those values, if there are any, or empty. In order to show how all this works, we consider the following quite typical operational situation:

"Extract from domain D_i those values that correspond in the domain D_j to the value K (assuming that an index has been built for D_j)."

This can be considered as a complex command which expands into the two elementary ones

1. T ← GET INDEX (D_j, K, '=') for read

2. V ← GET DOMAIN (D_i, T) for read

Step 1 performs the descent on the B-tree; it will also issue to the physical level the sequence of page calls to get the appropriate pages of the inversion of the domain D_j. The result returned in T is the set of tuple identifiers to which correspond those values of D_j equal to K. Step 2, using the tilac of D_i, converts each value in T into two values: logical page address and offset (i.e., the displacement in that particular page of the item corresponding to that value of T):

T = >[LPA, offset]

Then the sequence of commands for getting the pages whose addresses are in LPA's is issued to the physical level.

The function GET DOMAIN also groups together all those tid's that refer to the same page in the segment of the relation on hand, so that the relevant pages are read only once. The result is reordered according to the initial tid sequence. Finally, each page is processed to extract the values corresponding to the computed offsets and to build the resulting APL array.

The physical level represents the storage component of the system. It materializes the representation of the objects available at the upper level, accesses physical records from auxiliary storage, and controls their transfer to and from main storage.

The interface between the internal and the physical level is established by a "command language" which provides command functions to manipulate pages; that is, informally,

GET (page *i*, segment *j*) for read
GET (page *i*, segment *j*) for write

These functions always return as a result the page *i* or an error code. Here segment and page identifications are used to compute physical access paths to data and to records on data sets by issuing the access method commands.

Other important functions performed at the physical level are those of granting compatibility among lock requests by different transactions, providing mechanisms for assuring physical data base integrity, recovery schemata, and suitable sequencing of record access to reduce accessing time to a page. The description of all these issues is not covered here since they are beyond the scope of this paper.

Appendix B: The function *RESDUR*

Following is the definition of the function *RESDUR* to compute the residual duration of a given project *P*.

```
∇ RD←RESDUR P;D;PS;PN;DNS;
                    DS;RD1
[1]     RD←0
```

Set up the result.

```
[2]     →0 IF EMPTY P
```

Quit if *P* is empty.

```
[3]     D←DATEDUR P
```

D is the duration of *P*. The global variable *SDATE* (start date) is produced as a side effect by the function *DATEDUR* and contains the start date of each project.

```
[4]     PS←(SDATE≠0) SELECT P
```

The projects started have *SDATE*≠0.

```
[5]     PN←(SDATE=0) SELECT P
```

PN are projects not yet started.

```
[6]     →ALLSTARTED IF EMPTY PN
```

Go to *ALLSTARTED* if all projects are started.

```
[7]     ELSE:DNS←(SDATE=0) SELECT
                              DURATION
```

Else, take the duration of those not yet started.

```
[8]     RD←DATENORM RD+DNS+MAX
                        RESDUR SONS PN
```

The new residual duration is computed by adding the old one to the duration of projects not started and to the largest residual duration of the "sons" of the current project (in a recursive fashion). *DATENORM* provides for "normalizing" the result of this sum by imposing the number of months not to exceed 12 and the number of days not to exceed the number of days of that month.

```
[9]     →0 IF EMPTY PS
```

Quit if no project has started.

```
[10]    ALLSTARTED:DS←(SDATE≠0)
                        SELECT DURATION
```

Select duration of started projects.

```
[11]    RD1←(SDATE+DS)-TODAYDATE
```

Residual duration is endpoint minus today's date.

```
[12]    RD←DATENORM MAX (RD1>0)
                              SELECT RD1
```

The residual duration is the maximum duration among the activities still "alive."

```
∇
```

References

1. E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Commun. ACM* **13,** 377 (1970).

2. E. F. Codd, "Relational Completeness of Data Base Sublanguages," *Data Base Systems*, Vol. 6, Courant Computer Science Symposia Series, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1972.
3. E. F. Codd, "Further Normalization of the Data Base Relational Model," *Data Base Systems*, Vol. 6, Courant Computer Science Symposia Series, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1972.
4. M. M. Zloof, "Query by Example," *Proceedings National Computer Conference* (AFIP Press) **44,** 431 (1975).
5. D. D. Chamberlin and R. F. Boyce, "SEQUEL: A Structured English Query Language," *Proceedings 1974 ACM SIGFIDET Workshop*, Ann Arbor, MI, April 1974, pp. 249–264.
6. R. F. Boyce, D. D. Chamberlin, W. F. King, and M. M. Hammer, "Specifying Queries as Relational Expressions: The SQUARE Data Sublanguage," *Commun. ACM* **18,** 621 (1975).
7. G. D. Held, M. R. Stonebraker, and L. Wong, "INGRES: A Relational Data Base System," *Proceedings of the AFIPS National Computer Conference*, Anaheim, CA, May 1975.
8. *APL Language*, Report No. GC26-3847, IBM Corporation, White Plains, NY.
9. *APL Shared Variables (APLSV) User's Guide*, Report No. SH20-1460-1, IBM Corporation, White Plains, NY.
10. *APL Shared Variables (APLSV) Programming RPQ WE 1191 TSIO Program Reference Manual*, Report No. SH20-1463, IBM Corporation, White Plains, NY.
11. D. Gries, *Compiler Construction for Digital Computers*, John Wiley & Sons, Inc., New York, 1971, pp. 32–34.
12. D. D. Chamberlin, J. N. Gray, and I. L. Traiger, "Views, Authorization, and Locking in a Relational Data Base System," *Proceedings of the AFIPS National Computer Conference*, Anaheim, CA, May 1975.
13. P. Paolini and G. Pelagatti, "Formal Definition of Mappings in a Data Base," *Proceedings of the SIGMOD International Conference on Management of Data*, Toronto, Canada, August 1977, pp. 40–46.
14. ANSI/X3/SPARC Study Group on Data Base Management Systems, "Interim Report," *FDT Bull. ACM*, SIGMOD, 7, No. 2 (1975).
15. M. E. Senko, "DIAM as a Detailed Example of the ANSI-SPARC Architecture," *Proceedings of the IFIP Working Conference on Modelling in Data Base Management*, Freudenstadt, Germany, North-Holland Publishing Company, Amsterdam, 1976.
16. D. Bjorner, E. F. Codd, K. L. Deckert, and I. L. Traiger, "The Gamma-0 *N*-ary Relational Data Base Interface: Specifications of Objects and Operations," *Research Report RJ 1200*, IBM Research Laboratory, San Jose, CA, April 1973.
17. D. Knuth, *The Art of Computer Programming*, Vol. 3, *Sorting and Searching*, Addison-Wesley Publishing Co., Reading, MA, 1973, pp. 473–479.

Received October 11, 1977; revised April 26, 1978

The authors are located at the IBM Centro di Ricerca, Via Cardassi 3, 70121 Bari, Italy.