ARIEL -- A Semantic Front-End to Relational DBMSs

Robert M. Mac Gregor

System Development Corp., Santa Monica, CA 90406

This paper introduces the query language ARIEL, a language which retains the formal precision of relational languages such as SQL and QUEL, while exploiting the greater expressiveness of a semantic data model. ARIEL has been implemented as a front-end query language to several relational database systems, including to a front-end distributed DBMS being developed at SDC.

The most noteworthy elements of ARIEL are (1) a flexible syntax for expressing subqueries, (2) a convenient way to express "outer-joins" within a non-procedural framework (without the use of "null values"), (3) a comprehensive set of rules for defining the semantics of "reference chains", (4) a "light-weight" view mechanism, and (5) a clean semantics for expressing aggregate functions, especially in combination with the "group by" operator.

0. Introduction

ARIEL (A RetrIEval Language) is a calculus-based query language based on a semantic data model. The fundamental structure of ARIEL borrows heavily both from the relational query languages SQL [Date81, Chamberlin76], and QUEL [Stonebraker76, Ingres79] (familiarity with one or both of these languages is assumed in this paper). ARIEL was designed with the intent of allowing queries to be expressed more simply and clearly than is possible in relational languages such as SQL or QUEL. It seeks to achieve this end by (1) exploiting the possibilities of a semantic data model, and (2) increasing the clarity and orthogonality of some of the constructs which already exist in these other two languages (see [Date84]).

ARIEL invites comparison with two other semantic model-based query languages -- DAPLEX [Shipman81, Fox84] and GEM [Zaniolo83, Tsur84]. ARIEL is a non-procedural language, while DAPLEX is procedural -- thus, even though both languages are based on similar semantic models, they differ in their underlying approach. GEM, being firmly based on QUEL, can readily be compared with ARIEL. The philosophical basis of these two languages is different, however. GEM's designer deliberately avoided straying too far from the relational model, and as a result did not exploit constructs such as "set-valued attributes" to the fullest degree possible. ARIEL, on the other hand, is fully-committed to a semantic model base, and tries to exploit the possibilities stemming from the additional semantics as much as possible. Both GEM and ARIEL have been implemented to interface with relational database systems.

A translator is operational which translates ARIEL statements into QUEL (for an RTI Ingres DBMS), IDL (for a Britton-Lee IDM-500), SQL (for a Mistress DBMS), or DIL -- a language recognized by SDC's Mermaid system [TEMP83] (Mermaid provides the capability for distributed data management, acting as a front-end to multiple DBMS's connected via a network). Translating ARIEL into DIL is particularly challenging, since the DIL syntax does not permit subqueries, and hence a nested ARIEL query must be converted into a sequence of "flat" DIL queries.

The schema which we will use as a basis for our semantic model-based queries is illustrated in Appendix A. The model we are using is referred to as the "navigation" model. Structurally it is fairly similar to DAPLEX [Shipman81], but our choice of terminology is somewhat different. Readers familiar with semantic networks or semantic data models are likely to find the schema self-explanatory. Appendix B illustrates an "equivalent" schema phrased using relational model terminology. This schema will be referenced by queries which illustrate relational semantics.

The remaining sections of this paper treat the following topics: Section 1 presents the basic structure of ARIEL without going into any depth. Section 2 is primarily occupied with introducing ARIEL's "outer-join" construct. Section 3 defines the semantics of ARIEL's "reference chains". Section 4 introduces the **define** statement. Section 5 discusses aggregate functions in ARIEL, and Section 6 examines the semantics of ARIEL's **group by** clause. Section 7 briefly touches on several issues relating to the implementation of an ARIEL translator as a front-end to a relational DBMS. Finally, Section 8 contains some concluding remarks.

1. The ARIEL QUERY LANGUAGE

This section introduces the basic elements of the ARIEL query language, along with some terminology, and includes brief illustrations of some of its non-standard operators.

ARIEL's basic structure is fairly standard. The major structural unit is the *query block*, which consists of a *target list*, which specifies the columns to be output by that block, and three optional clauses: a *group-by* clause, a *qualification* (**where** clause), and an *order-by* clause. Below is the ARIEL formulation of the query "Retrieve names of employees whose age is greater than 50, sorted by their ages."

retrieve Name **of** Employee
  where Age > 50
  order by Age

Example 1.1

(Note: ARIEL supports the traditional "dot" notation (e.g., "Employee.Name"), but prefers an "of" notation in which the order of references is reversed). "Employee" in the above query is termed an *iteration variable* (sometimes called a "query variable"). Each iteration variable is bound to a Class over which we "iterate". The phrases "Name of Employee" and "Age" are termed *column references*.

The wildcard specification **all**, attached to an iteration variable, expands into a list of all Roles of the corresponding Class which (1) are SubComponents, and (2) are single-valued. If an iteration variable appears in a *target list* without a Role attached, a *default* Role is intended. The default Role of the Class Employee is the Role "Name" -- hence, the query "retrieve Employee" expands to "retrieve Name of Employee". An isolated iteration variable residing in other than a target list evaluates to a Role which serves as a "key" for instances of the corresponding Class. (Instances of this situation are illustrated further on).

ARIEL supports a test for equality of "enumerated" values using the operator **is**. The left-hand argument to **is** must reference a Role, while the right-hand argument is an enumerated value. For example:

retrieve Name **of** Employee
  where Sex **is** Female

Example 1.2

Because we have restricted enumerated values to appear only as right-hand arguments to the **is** operator, the Datatype for each value can be determined simply by inspecting the corresponding left-hand operand. As a consequence, the same label can be employed in more than one enumerated Datatype, and can also be used as a Role Name, without causing any ambiguities.

The **is** operator can also be used to specify subclass restrictions. In this case the left-hand argument is an iteration variable, and the right-hand argument is the name of a (sub)Class, e.g.,

retrieve Name **of** Dept
  where Dept **is** ManufacturingDept **or**
    Dept **is** EngineeringDept

Example 1.3

ARIEL supports the quantification predicates **some**, **all**, and **no**, using constructs analogous to those defined for DAPLEX, e.g.,

retrieve Name **of** Dept
  where
    all Employee
      where DeptId **of** Employee = DeptId **of** Dept
      have Age < 35

Example 1.4

In general, the predicate **all** (or **some** or **no**) is followed by a list of iteration variables. The **where** clause belonging to the **all** predicate is optional (and unnecessary for the **some** and **no** predicates). We prefer these calculus-style predicates to algebraic operators and predicates such as set-difference and set-containment (**contains**), although **contains** is useful enough that it may be added at a later date. Explicit specification of "join" predicates, as illustrated in Example 1.5, is seldom necessary in ARIEL. The "reference chain" construct defined in Section 1.3 will provide the following more attractive equivalent to the preceding query:

retrieve Name **of** Dept
  where all Employees **of** Dept **have** Age < 35

Example 1.5

2. Subqueries and Outer Joins

A *subquery* in ARIEL introduces a new scope inside of an ARIEL query, analogous to a nested **select-from-where** block in SQL. ARIEL subqueries appear in several forms. If an ARIEL expression is qualified by a **where** clause, and if the expression and its qualification are surrounded by parentheses, then the parenthesized expression represents a subquery. Secondly, expressions quantified by **all**, **some**, or **no** represent subqueries. Thirdly, the argument to an **in** operator (set membership) is always a subquery. Subqueries also appear as arguments to aggregates, and within "reference chains". Within the scope of the subquery of Example 2.1 below, "OtherEmp" names a *local variable*, "Employee" names an *external variable*, and "Salary of Employee" is an *external column reference* (the **define** statement is discussed in section 4). The query is "Retrieve names of employees whose salary is the same as that of some other employee":

define OtherEmp **is** Employee
retrieve Name **of** Employee
  where some OtherEmp
    has Salary = Salary **of** Employee

Example 2.1

Novel semantics result when a subquery appears within the target list of query. A property commonly attributed to subqueries is that they have no side-effects. A useful application of this property is illustrated in the following

example (assume "OtherEmp" is defined as before). The query is "Retrieve the name of each employee, together with a list of names of all employees having the same salary":

```
retrieve Name of Employee,
         Other := ( Name of OtherEmp where
                    Salary of OtherEmp = Salary of Emp)
```

Example 2.2

The first column of the result of this query contains the "Name" of each instance of "Employee". Associated with each "Employee" instance, in the second column, will appear zero, one, or several "Names" of those instances of "OtherEmp" which join successfully with the "Employee" instance. ARIEL adopts a hierarchic format to print the value of columns which evaluate to a set of values. Hence, a portion of the result looks like

Name	Other
Newton	Euclid
	Hilbert
	Leibnitz
Gauss	
Euclid	Newton
	Hilbert
	Leibnitz

Example 2.3

We observe that the join over the "Salary" columns of "Employee" and "OtherEmp" is actually an "asymmetric outer-join" [Date83]. Hence, placing subqueries in the target list results in a natural way to define (asymmetric) outer-joins. The expressive capability of this construct is equivalent to what can be expressed using the "nested" loops" construct commonly found in procedural query languages. A nested loops-style query (omitting formatting specifications) would look something like:

```
foreach Employee do
  begin
  print( Name of Employee);
  foreach OtherEmp is Employee do
    if (Salary of OtherEmp = Salary of Employee)
    then  print( Name of OtherEmp);
  end;
```

Example 2.4

The ARIEL query is noticeably more concise than the procedural version. In terms of ease of expression, ARIEL's outer-join construct also compares favorably with the outer-join constructs suggested by Chamberlin [80] and Date [83]. We consider it advantageous that ARIEL's outer-join construct does *not* require an understanding of *null values*. We note that, unlike these two latter outer-join proposals, ARIEL cannot express a symmetric outer-join. This could be remedied by adding a set-union operator to the ARIEL language.

We pause here to note an important aspect of our subquery construct which has a bearing on the semantics we assign to "reference chains" (the subject of the next section). Suppose we remove the parentheses from the query in Example 2.2. The result is

```
retrieve Name of Employee, Name of OtherEmp
  where Salary of OtherEmp = Salary of Emp
```

Example 2.5

Removing the parentheses has caused the subquery to evaporate, and the interpretation here is that the join is in fact an "equi-join". The query's meaning is "Retrieve names of all pairs of employees who have the same salary".

3. Reference Chains

ARIEL's most visible departure from relational query languages is its "reference chain" construct. A *reference chain*, written "Rk **of** Rk-1 **of** ... **of** R1" (equivalently, written "R1.R2. ... .RK" using "dot notation"), consists of a sequence of one or more *references* "Ri". In the simplest case, "R1" is the name of a Class, while R2 through Rk are the names of Roles which designate a directed path in the navigation model whose origin is the Class vertex to which R1 is bound. Here is an example query using reference chains "For each department whose department head has age greater than 50, print the department name and the name of that department head":

```
retrieve Name of Dept, Name of DeptHead of Dept
  where Age of DeptHead > 50
```

Example 3.1a

The reference "DeptHead" in the middle of the reference chain "Name of DeptHead of Dept" names a Role of the Class "Dept". This Role has Type Employee; hence "Name of DeptHead" and "Age of DeptHead" refer to Roles of Employee. Here is an equivalent query using relational semantics:

```
define HeadOfDept is Employee
retrieve Name of Dept, Name of HeadOfDept
  where Age of HeadOfDept > 50 and
        DeptHead of Dept = EmpId of HeadOfDept
```

Example 3.1b

The presence of the relational join in Example 3.1b is implied in Example 3.1a by the use of the Role "DeptHead". Also, note in Example 3.1a that the references to "DeptHead" in the chains "Age of DeptHead" and "Name of DeptHead of Dept" both refer to the same "variable". We will now examine the semantics of ARIEL's reference chains.

As a default, a reference is interpreted to have the same meaning everywhere within its scope (references in ARIEL follow block-structured scoping rules similar to those in SQL). For example, suppose a reference chain "... B of C" appears in the same block as (or in an outer block of) a reference chain "A of B". Then the "B" in "A of B" is interpreted to be equivalent to the "B" in "B of C", implying that "A of B" is really an abbreviation for "A of B of C". Using this rule we find that the following two queries are equivalent to the query of Example 3.1:

```
retrieve Name of Dept, Name of DeptHead of Dept
  where Age of DeptHead of Dept > 50
```

retrieve Name **of** Dept, Name **of** DeptHead
where Age **of** DeptHead **of** Dept > 50

Example 3.1c

Sometimes the syntax makes it clear that two references with the same name refer to separate entities. For example, if reference chains "Name of Dept" and "Name of Employee" occur in the same block, the two references named "Name" are not equivalent. In some cases, the context of a reference causes its meaning to be ambiguous; the following query is illegal because the chain "Name of DeptHead" could mean either "Name of DeptHead of Dept" or "Name of DeptHead of otherDept": "Retrieve names of persons who head more than one department"

define otherDept **is** Dept
retrieve unique Name **of** DeptHead
where DeptHead **of** Dept = DeptHead **of** otherDept

Example 3.2 (illegal)

References which are bound directly to Classes, such as the references "Dept" and "otherDept" in Example 3.2, are called *unqualified*. On the other hand, both references to "DeptHead" in Example 3.1a are said to be *qualified*, because their meaning depends upon their interpretation as a Role of some other reference (in this case "Dept"). A reference whose Type is an Entity-Class names a *variable*. In Example 3.1a, Dept is a variable ranging over the Class Dept, while DeptHead is a variable ranging over the Class Employee.

Thus far we have illustrated examples of qualified variables which are *explicitly qualified*: a variable R is explicitly qualified if R appears within at least one reference chain of the form "(... of) R of S (of ...)", i.e., in a chain where R is not the right-most reference. Variables can also be *implicitly qualified*. For example, in the query "Retrieve names of departments whose department heads are named 'Irving'":

retrieve Name **of** Dept
where Name **of** DeptHead = "Irving"

Example 3.3

the ARIEL translator determines that "DeptHead" is a valid Role of the Class "Dept", so it automatically expands "Name of DeptHead" to "Name of DeptHead of Dept".

We now take a closer look at the problem of determining the Class to which a variable in an ARIEL query is bound. Suppose the reference chain "Name of Employees" occurs within a query. In order to determine the Class associated with the reference "Employees", the ARIEL translator first determines whether or not "Employees" is within the scope of a chain such as "... Employees of Dept" which explicitly qualifies it. If this is not the case, the translator next looks for another variable bound to a Class such as Dept to which "Employees" can be qualified implicitly. If both of these conditions fail, the translator checks to see if "Employees" is the name of a Class (or is a **define** image -- see section 4). Summarizing, the rules used to bind a variable R are:

(1) Determine if R is explicitly qualified.
(2) If (1) fails, see if R is implicitly qualified.
(3) If (1) and (2) fail, see if R is the name of a Class (or image).

Some examples will help to motivate this choice of rules. Consider the following query "Retrieve names of department heads whose names start with 'S':

retrieve Name **of** DeptHead **of** Dept
where Name = 'S*'

Example 3.4

Although the reference "Name" in the qualification binds implicitly both to "DeptHead" and "Dept", there is no ambiguity because Rule (1) supercedes Rule (2), binding "Name" to "DeptHead". Next, consider the query

retrieve Employee, Dept **of** Employee, DeptHead **of** Dept

Example 3.5

Because Rule (1) supercedes Rule (3), "DeptHead of Dept" is equivalent to "DeptHead of Dept of Employee", i.e., both references to "Dept" refer to the same variable. Applying Rule (3) rather than Rule (1) would have created two separate variables named "Dept", resulting in a cartesian product.

When nested query blocks are involved, the first three rules are governed by an additional rule. Consider the query "Retrieve names of departments having more than one employee named Irving":

retrieve Name **of** Dept
where count(Employees **where** Name = 'Irving') > 1

Example 3.6

We consider it desirable that the reference "Name" in the predicate "Name = 'Irving'" bind (implicitly) to "Employees" rather than (explicity) to "Dept". Hence we add

(2.5) Apply Rules (1) and (2) to each query block before considering the next outer query block. If they fail to apply at all levels, try Rule (3).

As an example of how Rule (2.5) limits the application of Rule (3), suppose that "Employees" was the name of a Class as well as a Role of the Class Dept. Rule (2.5) is phrased so that the reference "Employees" in Example 3.6 would still be bound to (implicitly qualified by) "Dept", rather than being defined as unqualified.

Allowing variables to be qualified *implicitly* opens the door to some subtleties which will almost surely escape the casual user. Consider the classic query "For each employee who earns more than their manager, print the employee's name and the manager's name":

retrieve Name **of** Employee, Name **of** Manager
where Salary **of** Employee > Salary **of** Manager

Example 3.7

Rule (2) expands "Name of Manager" to "Name of Manager of Employee". A second application of Rule (2) faces a problem: "Salary of Manager" can expand to "Salary of Manager of Employee" or "Salary of Manager of Manager of Employee". We are saved from an ambiguous interpretation by recalling the default rule that a reference is interpreted to have the same meaning everywhere, which in this case implies that the

two occurences of the variable "Manager" should have identical qualifications. In general, allowing variables to be implicitly qualified adds a bit of user-friendliness, but increases the possibility that a query will be ambiguous. Because of this, allowing an ARIEL query to contain implicitly qualified variables is an option which can be disabled if the user so wishes. In some cases, the specification of a cartesian product can happen *only* if implicit qualification is turned off. For example, ARIEL implicitly links the Classes Employee and Dept in the ARIEL query **"retrieve** Employee, Dept".

For each qualified variable in a reference chain there corresponds a join predicate relating that variable to the one that qualifies it. Depending on the context of the qualified variable, that join is interpreted either as an outer-join or an equi-join. Qualified variables which only appear in a target list are linked by outer-joins. For example, the query "For each department, print its name and the names of any of its employees":

retrieve Name **of** Dept, Name **of** Employees **of** Dept

Example 3.8a

is equivalent to the relational query

retrieve Name **of** Dept,
(Name **of** Employee **where** DeptId **of** Employee = Dept)

Example 3.8b

(the semantics of a subquery which appears in a target list is described in Section 2).

Qualified variables which appear anywhere other than a target list are linked by equi-joins. Consider the query "Foreach department having at least one employee over age 50, print its name, and the names of any employees over age 50":

retrieve Name **of** Dept, Name **of** Employees **of** Dept
where Age **of** Employees > 50

Example 3.9a

The variable "Employees" occurs in both the target list and the qualification. Hence it is linked by an equi-join. An equivalent relational query would be

retrieve Name **of** Dept, Name **of** Employee
where DeptId **of** Employee = Dept **and**
Age **of** Employee > 50

Example 3.9b

If a user wanted to preserve outer-join semantics, resulting in the query "Foreach department, print its name and the names of any department employees over age 50", s/he could write

retrieve Name **of** Dept,
(Name **of** Employees **of** Dept
where Age **of** Employees > 50)

Example 3.9c

We note that in this example it is the subquery semantics, rather than the reference-chain semantics, which implies an outer-join.

Implementation Note: The ARIEL translator utilizes a post-processor to compute outer-joins, since none of the back-end DBMSs it translates to support an outer-join. The translator recognizes cases where an outer-join implied by a reference chain can be replaced by an equi-join without altering the meaning of the query. For example, the link for "DeptHead of Dept" can be represented by an equi-join rather than an outer-join if the navigation schema indicates that the Role DeptHead cannot have a null value.

4. Define Statements and Images

The basic **define** statement represents a fusion of the QUEL "range" statement with a view definition construct. An example of a define is:

define TopEmployee **is**
Employee **where** Salary > 50000

Example 4.1

"TopEmployee" represents the set of instances of employees whose salary exceeds 50000. In subsequent queries, "TopEmployee" can appear anywhere that "Employee" can. We refer to "TopEmployee" as an *image*. An image functions semantically exactly like a view, except that it exists only for the lifetime of a user session. Permanent images, which we call "views", are formed by including the keyword **view**:

define view TopEmployee
is Employee **where** Salary > 50000

Example 4.2

Using the keyword *snapshot* causes a snapshot of the image to be created:

define snapshot TopEmployee **is**
Employee **where** Salary > 50000

Example 4.3

The snapshot creates a new table in the database containing an evaluation of the image "TopEmployee". Snapshots are automatically deleted at the end of a user session.

It is intended that ordinary define statements be used liberally, as QUEL range statements are, rather than frugally, as views usually are. This is practical because an ARIEL image does not carry the administrative overhead that accompanies a view definition. Define statements can be nested, e.g.

define TopSalesman **is**
(TopEmployee **where**
Name **of** Dept **of** TopEmployee = "Sales")

Example 4.4

This "successive refinement" of image definitions can be used to facilitate *browsing* -- a user can define on-the-fly a collection of images which describe various useful subsets of data.

The semantic network underlying ARIEL introduces some aspects in the definition of images which are not applicable to a relational data model. Notice that in the target list of Example 4.1, "Employee" is used without naming a Role link. Used in this way within the target list of an ordinary ARIEL query, "Employee" would be interpreted as "Name of Employee" (some Role in each Class supplies the default value for this construct). This is not an acceptable interpretation within a **define** statement, since that would imply that "TopEmployee" has only the single Role "Name". Instead, "Employee" by itself in the target list of a **define** statement is interpreted to include both its SubComponent and Navigation Roles. (Notice the reference chain "Name of Dept of TopEmployee" in Example 4.4). We also note that if "TopEmployee" has instead been defined as

define TopEmployee **is**
Employee.all **where** Salary > 50000

Example 4.5

such a reference chain would be illegal, since an **all** qualifier expands only the SubComponent Roles of a Class.

An image definition whose target list consists of a single Class name (such as in Examples 4.1 and 4.4) defines a *subclass* of that Class. The usual rules of inheritance apply. ARIEL's semantic network adds an additional inheritance rule not commonly found in other networks.

Notice that the link from "Dept" to "Employee" is named "Employees". In the general case when a link 'T' runs *from* a Class 'F' *to* a Class (also named) 'T' (or 'T' suffixed with an 's' when 'T' is a set-valued link), each subclass 'S' of Class 'T' inherits the link 'T' but the induced link is renamed 'S' (suffixed by 's' if appropriate). In our example, the definition of the image/Class "TopEmployee" has the side-effect of inducing a new link named "TopEmployees" running from "Dept" to "TopEmployee". This means that a reference chain such as "Salary **of** TopEmployees **of** Dept" is valid.

The effect of this rule is that new classes introduced by the define mechanism are not necessarily isolated from the rest of the network. Instead, navigation paths to the new class are immediately available (whenever the conditions just described are satisfied).

Below is an example of the construct to define (virtual) links:

define Manager **of** Employee **is**
DeptHead **of** Dept **of** Employee

Example 4.6

This defines a new link named "Manager" emanating from the Class "Employee". We note that traditional view mechanisms do not provide the ability to add new links (columns) to existing Classes (tables). The expression following **is** can be a single- or set-valued query. Again, a virtual link defined by a **define** statement exists only for the lifetime of a user's session, unless the keyword **view** is included.

5. Aggregate Functions

When considering the way in which aggregate functions are defined in QUEL and SQL, we observe that each language has some advantages not enjoyed by the other. ARIEL's aggregate construct attempts to capture the advantages present in each of these languages. Our presentation of ARIEL aggregate functions will be accompanied by a comparison with the paradigms adopted by these other query languages.

QUEL aggregates have four notable advantages over their SQL counterparts: *First*, they are "purer" than SQL aggregates -- the argument to a QUEL aggregate must evaluate to a set. An SQL aggregate in some sense "coerces" the type of its operand. In example 5.1a below, "Employee.Salary" is single-valued,

select Employee.Salary
from Employee

Example 5.1a (SQL)

select max(Employee.Salary)
from Employee

Example 5.1b (SQL)

while in example 5.1b it evaluates to a set. *Second*, QUEL aggregates are more *orthogonal* [Date84] -- they can appear anywhere in both target-list and qualification expressions, whereas in SQL the use of aggregates in the qualification is restricted. *Third*, QUEL aggregates can be nested -- SQL's can't. And *finally*, QUEL aggregates capture an "outer-join" semantics which is not expressible in SQL, as exemplified by the QUEL query in Example 5.2 ("Retrieve the name of each department, and a count of the number of people in it"), which cannot be expressed in SQL with fewer than three statements (a "**select** into"; a "**delete**"; and a "**select**"):

range of e **is** Employee
range of d **is** Dept
retrieve (d.Name, **count**(e.Name **by** d.Id
where e.DeptId = d.Id))

Example 5.2 (QUEL)

ARIEL captures all of the advantages noted for QUEL aggregates by allowing aggregate functions to be wrapped around subqueries, just as they are in QUEL. In addition, because of its more powerful scope rules (borrowed from SQL) ARIEL can avoid employing the **by** clause needed in many QUEL queries. For example, the ARIEL equivalent of Example 5.2 would be

retrieve Name **of** Dept, **count** Employees **of** Dept

Example 5.3

(Note: In ARIEL, the argument of a **count** aggregate omits specification of the "final" value-link, since it has no semantic significance, e.g., we did not write "**count** Name of Employee of Dept" in Fig. 5.3).

In general, an aggregate operator can be applied to any *set-valued* ARIEL expression. AN aggregate induces its

argument to be a *subquery*, with accompanying implications regarding the scoping of variables. For example, if "PhoneNumbers" is a set-valued attribute of "Dept", then

```
retrieve Name of Dept, count PhoneNumbers
```

Example 5.4

is a legal ARIEL query. The expression "PhoneNumbers" is a *constant* within the subquery "**count** PhoneNumbers", but because it is set-valued, applying the aggregate operator makes perfect sense. Next, consider the query "Retrieve the values of the maximum and minimum salaries among all employees":

```
retrieve max Salary of Employee,
    min Salary of Employee
```

Example 5.5

Here, **max** and **min** define two subqueries, implying that the two references to "Employee" refer to two distinct iteration variables. However, if we qualify "Employee" in the following manner

```
retrieve max Salary of Employee,
         min Salary of Employee
 where Name of Dept of Employee = 'KBS'
```

Example 5.6 (illegal)

then "Employee" refers to a single iteration variable, bound to the outer-most query block. This implies that both expressions "Salary of Employee" in the two subqueries evaluate to single-valued constants. Hence, the query of Example 5.6 is *not legal.*

Before examining further aspects of ARIEL's aggregates, we will finish our earlier discussion by noting two advantages which SQL's aggregates have over QUEL's. *First*, for the case of aggregates within a *qualification*, SQL's scoping avoids the use of a **by** clause in the same way that it was avoided in the target-list in the ARIEL query of Example 5.3. This is illustrated by the following equivalent QUEL, SQL and ARIEL queries "Retrieve names of departments having more than 10 employees":

```
range of d is Dept
range of e is Employee
retrieve d.Name
 where count( e.Id by d.Id
                    where e.DeptId = d.Id ) > 10
```

```
select Dept.Name
from  Dept
where (select count(*)
       from  Employee
       where Employee.DeptId = Dept.Id) > 10
```

```
retrieve Name of Dept
 where count Employees of Dept > 10
```

Example 5.7

Second, for queries containing what might be termed a "replicated qualification", SQL can often phrase queries much more concisely than is possible in QUEL. As an example, consider the following equivalent SQL and QUEL queries "Retrieve the maximum and minimum salaries among all employees in the KBS department":

```
select max( Employee.Salary ), min( Employee.Salary )
from  Employee, Dept
where Employee.DeptId = Dept.Id and
      Dept.Name = 'KBS'
```

```
retrieve (
  max( Salary of Employee
       where Name of Dept of Employee = "KBS"),
  min( Salary of Employee
       where Name of Dept of Employee = "KBS") )
```

Example 5.8a

ARIEL's syntax for aggregates is closer to QUEL's than SQL's. (Note that the illegal ARIEL query of Example 5.6 has the same syntactic form as the above SQL query). However, ARIEL can still achieve roughly the same degree of conciseness as SQL by exploiting the power of its **define** statement, as illustrated by Example 5.8b:

```
define KBSsalary is Salary of Employee
     where Name of Dept of Employee = "KBS"
retrieve max( KBSsalary ), min( KBSsalary )
```

Example 5.8b

Thus, we see that by straightforward means, or by ones that are only slightly devious, ARIEL can achieve all of the advantages of QUEL's aggregates, and also those advantages enjoyed by SQL. ARIEL will lend help to users who are confused by illegal queries such as the one in Example 5.6 -- the translator will print out a message suggesting that a **define** statement be used in this case.

6. The Group-By Clause

The **group by** clause in an ARIEL query block allows a user to specify a partition over the image referenced by a local iteration variable, in a manner similar (but not identical) to the SQL **group by**. ARIEL mandates that the columns listed in an ARIEL **group by** clause must all be bound to the same variable, called the *group-by variable* (this restriction, as we shall see, does not reduce ARIEL's expressive power). An additional restriction is that column references in a **group by** list must be single-valued. A **group by** clause produces the following important effect: The **group by** clause induces all column references bound to the group-by variable which are *not* listed in the **group by** clause to become *set-valued*. For example, in the query below ("For each department, print its department ID and the names of all employees in it") the reference "Name **of** Employee" is set-valued.

```
retrieve DeptId, Name of Employee
 group by DeptId
```

Example 6.1

This implies that the output of the query will be hierarchic, e.g.,

DeptId	Name
25	Newton
	Kepler
32	Gauss
	Hilbert
	Euclid

Example 6.2

This further implies that we can wrap an aggregate around any column reference bound to the group-by variable, but not listed in the **group by** clause, e.g., "For each department having more than 10 employees, print its department ID and the names of all employees in it":

retrieve DeptId, Name **of** Employee
  group by DeptId
  where count Employee > 10

Example 6.3

However, a third implication is that these latter column references cannot be used as arguments to operators which require single-valued arguments, e.g., in the query below, "Salary", as a set-valued variable, is incompatible with the '>' operator.

retrieve DeptId, **count** Employee
  group by DeptId
  where (Salary > 50000) **and** (**max** Salary < 100000)

Example 6.4 (illegal)

Furthermore, the presence of a **group by** clause precludes the participation of the group-by variable in any join conditions, except where the join column is a member of the **group by** list.

These difficulties are presumably what led the designers of SQL to split the SQL qualification into two parts, the **where** clause and the **having** clause. This allows one to write (in SQL):

select DeptId, **count** Employee.Id
from Employee
where Salary > 50000
group by DeptId
having **max**(Salary) < 100000

Example 6.5 (SQL)

(the query is "For each department for which at least one employee salary exceeds 50000, and for which all employees have salaries under 100000, print its department ID and the number of employees in it with salaries over 50000"). Within the SQL query of Example 6.5, the column reference "Salary" is interpreted as single-valued when it appears within the **where** clause, and as set-valued when it appears in the **having** clause. We suspect that this lack of uniformity is one of the reasons why SQL's **group by** and **having** clauses are relatively difficult for people to comprehend (the author made this observation while teaching SQL in graduate-level data management classes).

By utilizing ARIEL's **define** statement, we can construct an ARIEL query equivalent to that in Example 6.5.

define TopEmployee **is**
  Employee **where** Salary > 50000
retrieve DeptId, **count** TopEmployee
  group by DeptId
  where **max** Salary < 100000

Example 6.6

This trick always works -- for any SQL query which includes both **where** and **having** clauses, an equivalent ARIEL query is constructed by moving the contents of the SQL **where** clause into an ARIEL **define** statement, and then placing the contents of the SQL **having** clause into an ARIEL **where** clause. By use of this device, we can insure that column references are uniformly single- or set-valued across an ARIEL statement.

The second source of difficulty users have with the SQL **group by** is that they find it difficult to *visualize* the concept of a "partition over a restricted cross-product of relations". ARIEL's **group by** semantics require that a user place the restricted cross-product definition in a **define** statement, and then *give it a name*. This creates a new semantic entity over which grouping seems relatively straight-forward.

A final note on the **group by**: The existence of set-valued links in ARIEL largely does away with the need for a **group by** clause. The clause exists only to provide a reasonable way to group on *non-key* attributes (grouping on non-key attributes can be accomplished by adding extra variables and joining them on these attributes. This is an awkward solution which we don't recommend).

7. Notes on the Implementation of ARIEL

What we have referred to here as "the ARIEL translator" is actually a more general translator which can translate any one of three languages (ARIEL, SQL, or DIL) into any one of four languages(SQL, QUEL, IDL, and DIL). This translator is utilized in multiple places within a heterogeneous distributed DBMS front-end being developed at SDC.

To translate ARIEL queries, the translator utilizes a pair of internal schemas -- one defining the semantic model seen by the user, and one mirroring the (relational) schema of the target DBMS. A set of mapping tables guides the translation across schemas. For example, a table maps each entity-valued Role onto an equivalent join predicate.

DIL -- the relational language utilized by the Mermaid Distributed Query Optimizer [Templeton83], cannot express "nested" subqueries. Hence, ARIEL queries containing nested subqueries are "flattened" before being translated into DIL. For example, the query "Retrieve names of departments having no employees":

retrieve name **of** Dept
where count(Employees **of** Dept) = 0

Example 7.1a

is translated into a sequence of three actions:

retrieve DeptId, Name **of** Dept **into** Temp

delete Temp
where DeptId **of** Temp = DeptId **of** Employee

retrieve Name **of** Temp

Example 7.1b

This exemplifies one three different transformations which the translator employs to process aggregated subqueries. The most general of them is a variation on Epstein's algorithm for evaluation aggregates in QUEL [Epstein79]. We should mention that the flattening strategies described in [Kim82] are not suitable for our present application. Some of Kim's transformations employ an "anti-join" operator which is not supported in current relational query languages. Also, his transformation for handling aggregates cannot handle the case "count(...) = 0".

Data retrieved by the translator may be processed by a Post Processor module which has the capability to compute outer-joins. We have deliberately avoided null-value based strategies for computing outer-joins, such as the one described in [Tsur84], because we wish to avoid the necessity for modifying the original schema and data of the underlying database.

8. Conclusion

The ARIEL language introduces several innovations in the area of formal query languages. The semantic constructs, especially reference chains, result in queries which are much more English-like than their relational-query counterparts. We note that, although ARIEL is based on a semantic data model, many of its constructs (e.g., outer-join semantics, define statement, aggregate and group-by semantics) could be utilized in a strictly-relational query language.

The ARIEL translator demonstrates the feasibility of installing a semantic-model based front-end to *unmodified* relational databases. We anticipate that the presence of *links*, as embodied in reference chains, and of *set-valued attributes* will greatly facilitate planned future efforts to interface ARIEL to network or hierarchic databases.

Acknowledgement: I would like to thank Margaret Reid-Miller, who helped to clarify and simplify the semantics of ARIEL's reference chains. I also benefitted from numerous discussions with Dan Kogan, Marjorie Templeton, and Iris Kameny on various aspects of the ARIEL language. Finally, I wish to thank Stephen Russell, who typeset the final version of this paper.

REFERENCES

[Chamberlin76]
Chamberlin, D.D., et al., *SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control* IBM J. Res. Develop., 20,6, November, 1976 (pp. 560-574).

[Chamberlin80]
Chamberlin, D.D., *A Summary of User Experience with the SQL Data Sublanguage*, Proc. Int'l Conf. on Data Bases, Aberdeen, Scotland, July 1980. Also IBM Research Report RJ2767, San Jose, CA. April 1980.

[Chen76]
Chen, P.P., *The Entity-Relationship Model--Toward a Unified View of Data*, ACM Trans. Database Syst., 1,1, March 1976 (pp. 9-36).

[Date81]
Date, C. J., An Introduction to Database Systems, Vol. I (Third Edition), Addison-Wesley, 1981.

[Date83]
Date, C. J. An Introduction to Database Systems, Vol. II, Addison-Wesley, 1983.

[Date84]
Date, C. J., *Some Principles of Good Language Design*, ACM SIGMOD Record, 14,3, Nov., 1984.

[Fox84]
Fox, S., et. al., DAPLEX User's Manual, Computer Corporation of America, Cambridge, Massachusetts, 1984.

[Epstein79]
Epstein, Robert, *Techniques for Processing of Aggregates in Relational Database Systems*, Electronics Research Lab Report No. UCB/ERL-M79/8, University of Calif., Berkeley, CA, February 1979.

[Ingres79]
Woodfill, J., K., et al., *INGRES Version 6.2 Reference Manual*, Electronics Research Lab Report No. UCB/ERL-M78/43 University of Calif., Berkeley, CA, 1979.

[Kim 82]
Kim, Won, *On Optimizing an SQL-like Nested Query*, ACM Trans. Database Syst., 7,3, September 1982 (pp. 443-46).

[Shipman81]
Shipman, D.W., *The Functional Model and the Data Language DAPLEX*, ACM Trans. Database Syst., 6,1, March 1981 (pp. 140-173).

[Stonebraker76]
Stonebraker, M., E. Wong, and P. Kreps, *The Design and Implementation of INGRES*, ACM Trans. Database Syst., 1,3, September 1976 (pp. 189-222).

[Templeton83]
Templeton, M. et al, *An overview of the Mermaid System -- A Frontend to Heterogeneous Databases*, Proc. IEEE EASCON Conf., Washington, D.C., September, 1983.

[Tsur84]
Tsur, S., and C. Zaniolo, *An Implementation of Gem -- Supporting A Semantic Data Model on a Relational Back-End* Proc. ACM/SIGMOD Conf., SIGMOD Record 14,2, June, 1984.

[Zaniolo83]
Zaniolo, C., *The Database Language GEM*, Proc. ACM/SIGMOD Conf., SIGMOD Record 13,4, May, 1983.

Appendix A: Navigation Model Schema

The schema shown below is expressed in terms defined by a simplified version of a model of our own devising, termed the "navigation" model. The discussion following the schema is included primarily as a means of acquainting the reader with our choice of terminology.

```
class Employee
 roles
   Name       : s32 default,
   Age : i2,
   Sex : Sex,
   Salary     : i2,
   Dept: Dept
   ;

class Dept
 roles
   Name              : s20 default,
   Division          : s20,
   DeptHead          : Employee,
   Employees  : set of Employee,
   PhoneNumbers      : set of d7
   ;

class ManufacturingDept
 superclasses Dept
 ;

class EngineeringDept
 superclasses Dept
 roles
   TechnicalMgr      : Employee;
   ;

datatype Sex enum ( Male, Female );
```

The definitions define four "Classes": Employee, Dept, ManufacturingDept, and EngineeringDept, and one "Datatype": Sex. A *Class* is defined by listing its "SuperClasses" and "Roles". Each Class defined above has a single SuperClass (Employee and Dept SuperClasses default to the root Class).

A *Role* is defined by its "Name", "Type", and an optional list of other "facets". Roles listed have as their *Type* either a system-defined "Datatype" (e.g., d4, s32, i2), a user-defined Datatype (Sex), or a Class. The Type "set of d7" is filled by a set containing zero or more instances of 7-digit integers.

Appendix B: Relational Model Schema

The schema shown below is expressed using relational model terminology. It can be considered a relational "realization" of the Employee and Dept Classes defined in the navigation model schema of Appendix A.

```
relation Employee
 attributes
   EmpId      : d4,
   Name       : s32,
   Age : i2,
   Sex : c1,
   Salary     : i2,
   DeptId     : d4
   ;

relation Dept
 attributes
   DeptId     : d4,
   Name       : s20,
   Division   : s20,
   DeptHead   : d4
   ;

relation DeptPhoneNos
 attributes
   DeptId     : d4,
   Number     : d7
   ;
```

Appendix C: ARIEL Grammar

In the grammar below, the symbols {, }, [,] , * , | , and ::= are metasymbols. Symbols enclosed in single quotes denote themselves. {...} denotes a mandatory syntactic element, [...] denotes an optional element, and {...}* denotes zero or more occurrences of an element. ARIEL's grammar is relatively small because typing rules are enforced by semantic rather than syntactic means.

```
program ::=      { action | transaction }*

transaction ::= begin { action }* end

action ::=       queryAction | defineAction
         |       dmlAction | ddlAction

queryAction ::= { retrieve | print | append }
                   queryExpr

queryExpr ::=  targetList { clause }*

targetList ::=  [ intoClause ] [ unique ]
         |      targetItem { , targetItem }*

targetItem ::=  [ identifier := ] arithPrim

intoClause ::=  { into | to } table

clause ::=       where boolExpr | unique | intoClause
         |       [ group ] by column { , column }*
         |       order by orderItem { , orderItem }*

orderItem ::=   column [ asc | desc ]

boolExpr ::=     boolExpr { or | and } boolExpr
         |       not boolExpr | predicate

predicate ::=    arithExpr compareOp arithExpr
         |       quantifier quanList [ where boolExpr ]
                   { has | have } boolExpr
         |       column is identifier
         |       arithExpr between arithExpr
                   and arithExpr
         |       arithExpr

compareOp ::= in | = | < | > | != | < > | < = | > =

quantifier ::=  some | all | no

quanList ::=    quanItem { , quanItem }*

quanItem ::=    column | variable in column

arithExpr ::=    arithExpr { + | - | '*' | / } arithExpr
         |       - arithExpr | arithPrim

arithPrim ::=    column | literal | '{' literalList '}'
         |       ( queryExpr )
         |       aggOp [ unique ] aggOperand

column ::=       attribute | { arrow of }* variable
         |       variable { . arrow }*

arrow ::=        { attribute | all | '*' }

literal ::=      string | integer | float | null

aggOp ::=        set | max | min | avg | count | sum
         |         identifier

aggOperand ::= ( targetList [ where boolExpr ] )
         |       column

defineAction ::=  define variable { is | isa } queryExpr

dmlAction ::=    insert [ into | to ] table
                   [ columnSpec ] tuple { , tuple }*
         |       { update | replace } table [ set ]
                   targetList [ where boolExpr ]
         |       delete [ from ] table
                   [ where boolExpr ]
         |       add arithExpr [ where boolExpr ]
                   to column [ where boolExpr ]
         |       remove arithExpr [ where boolExpr ]
                   from column [ where boolExpr ]

columnSpec ::= ( attribute { , attribute }* )

tuple ::=        < literal { , literal }* >

ddlAction ::=    create [ table ] tableVar
                   ( attribDefn { , attribDefn }* )
         |       destroy [ table ] tableVar
                   { , tableVar }*

tableVar ::=     variable | table

attribDefn ::=  attribute : class

variable ::=     identifier

table ::=        identifier

attribute ::=    identifier

class ::=        identifier
```