A Relational Interface to an Object Based System
or
Translating SQL to ADAMS

Thomas P. Cleary

IPC-TR-91009
August 8, 1991

Institute for Parallel Computation
School of Engineering and Applied Science
University of Virginia
Charlottesville, VA 22901

This research was supported in part by JPL Constract
#957721 and DOE Grant #DE-FG05-88ER25063.

Abstract

The evolution of database technology has resulted in the widespread use of three major database models. These three most popular models are: the hierarchical model, the network model and the relational model. The existence of numerous databases and database applications has given rise to a strong desire for the ability to use the data and applications created using one database model by other database systems.

The project that this report documents demonstrates that a relational database system can be mapped onto an object based system. In particular, a lex-yacc based translator was developed to convert the Structured Query Language (SQL) into the Advanced Data Management System (ADAMS) language. The basic relational SQL data definition, manipulation and query statements are shown to be translatable into equivalent ADAMS statements. The translated code can then be executed to emulate the actions SQL would have performed. This demonstrates that ADAMS can provide low-level implementation support to the relational database model

Acknowledgments

I would like to thank my advisor, Prof. J. L. Pfaltz, and Prof. J. French for their help with this project.

Table of Contents

1 Introduction

The evolution of database technology has resulted in widespread use of three major database models. These three most popular models are: the hierarchical model, the network model and the relational model. New database models such as object-oriented databases continue to emerge. The existence of numerous databases and database applications using the different models has given rise to a strong desire for the ability to use the data and applications created using one database model by other database systems.

The designers of the object based Advanced Data Management System (ADAMS) assert that ADAMS is sufficiently flexible and powerful enough to support the most popular database models. Although, it is the least complex of the major database models, the relational model is the dominant database model; therefore it was selected as the target model for investigation into this claim.

This project demonstrates that a relational database system can be mapped onto an object based database system. In particular, a translator was developed to convert the Structured Query Language (SQL) into the Advanced Data Management System (ADAMS) language. The BNF grammar as implemented SQL syntax is given in appendix A. The yacc version of the grammar is included in appendix B. The basic relational SQL data definition, manipulation and query languages are shown to be translatable into the object-oriented ADAMS language. The translated code can then be executed to emulate the actions SQL would have performed. This conclusively demonstrates that ADAMS can support the relational database model.

This document provides an overview of both SQL and ADAMS, presents the model used to map a relational database onto an object oriented database, details the translation of SQL statements into executable ADAMS code, and provides an example of a working translated database.

2 Introduction to the Relational Database Model

Since its introduction in 1970 the Relational database model has enjoyed considerable success. This is the result of both its simplicity and its theoretical foundations which facilitate analysis of formal relational database theory.

2.1 The Relational Model

A relational database collects the data into relations. Informally, a relation is a table consisting of rows and columns of related data. The columns, called attributes, are named and contain data values which are all of the same type. The type, or range of values that a particular attribute can assume is its domain. The rows, called tuples, are collections of attribute values.

A table or relation has the following properties:

- Each row is an n-tuple of the n attributes of the relation.
- The relation is a set of rows. Therefore ordering of the rows is unimportant.
- A row is an ordered list of values, order is significant. However this is only to correspond the values with the attribute name. If the values are linked to the attribute in some other way, order is immaterial.

A great appeal of the relational database model is that is does have a formal mathematical foundation. A relation in a relational database is a relation in the mathematical sense. Codd [Cod70] originally defined a relation by: given sets $D_1, D_2,...,D_n$, $\boldsymbol{R}$ is a relation on these n sets if it is a subset of the Cartesian product $(D_1 \times D_2 \times \ldots \times D_n)$. The set D_j is the *j*th *domain* of $\boldsymbol{R}$.

Thus a relation is a set of n-tuples over the domains S_j. The n-tuples themselves may be viewed as sets of n objects, one from each of the n domains. This is important for the translation into ADAMS as will be seen.

The set of attributes in a relation is the relation schema. The relation schema can be regarded as a template for a relation. These definitions correspond to the ADAMS language constructions and are given here for convenience.

2.2 The Structured Query Language

There are several commercially available languages which are based on the relational database model. Some of the most popular are the Structured Query Language (SQL), Query-By-Example (QBE) and Quel. SQL seems to be the most prevalent; therefore it was the language chosen to be implemented over ADAMS.

SQL previously SEQUEL, *s*tructured *e*nglish *que*ry *l*anguage is a descriptive rather than a procedural language[1]. It includes features to define a database, update the database and to query the database. Thus it is both a data definition and data manipulation language as well as a query language. It is a fairly simple, user friendly interface to a relational database.

In SQL the attributes, tuples and relations are called fields, rows and tables respectively. The equivalent terms will be used interchangeably in this document. An SQL database consists of one or more tables.

The basic data definition construct is the **create table** statement This statement specifies the name of the table and the fields and types for each field. An example is:

```
create table table_name
    (
    field1      type1,
    field2      type2,
    ...
    fieldn      typen
    );
```

Data manipulation occurs through the use of the **insert**, **delete** and **dump** statements. Chapter 5 on translating SQL's data manipulation language will present these statements in detail.

Queries consist of three clauses, the *select* clause, the *from* clause and the *where* clause. The *select* clause indicates which fields should be selected and, unless it is clear

1. SQL is roughly based on a formal query language, the tuple calculus.

from the context, from which relation the field is to be selected. The *from* clause lists all of the relations that are used in the query. The *where* clause establishes selection criteria for the rows. It is the true “query”. Actually it is optional and may be omitted to permit a projection in the relational algebra sense or to perform the Cartesian product of relations. Summarizing, a select statement has the form:

```
select field_list
from   relation_list
where  expression
```

The syntax of the select statement will be described in more detail in Chapter 6 and many examples of select statements will be presented throughout this document.

SQL supports many more statements than have been presented but the statements discussed are sufficient to support a relational database. Appendix A is a complete BNF grammar of the SQL syntax which has been implemented in the SQL to ADAMS translator.

3 Introduction to ADAMS[2]

ADAMS, the Advanced Data Management System is not a database system in the traditional sense. Rather it provides a clean interface to a large virtual persistent memory which users may access much as current programming languages allow access of data in volatile memory. The ADAMS language is a tool for defining, naming and accessing the persistent database structures. User programs are written in an existing language, such as C or Fortran (the host language), with ADAMS language statements embedded within the host language much as SQL in often embedded in a host language. Again much as in SQL systems, e.g. DB2, there exists a preprocessor which translates the embedded ADAMS statements into the host language with function calls to the ADAMS run time system. These functions provide access to the ADAMS dictionary which is the mechanism that references the persistent name space, and access to the storage manager which manages the actual data stored by the system.

The user of the ADAMS system creates a database which is structured as the user desires from the basic ADAMS constructs. These basic constructs mirror the object oriented ideas of classes and inheritance among classes.The definition of object oriented is not concrete, but ADAMS is an object oriented or at least an object based system. In ADAMS methods acting on objects are not provided. By the criteria of [Weg87] then, ADAMS would be called object based.

There are only a few primitive ADAMS constructs but they are sufficient to support most database models. The primitive constructs with a brief explanation are:

Element: An element is an identifiable entity. Elements may be regarded as the objects which make up the database. This is not actually a construct, but rather a concept necessary for the definitions which follow.

Codomain: A regular set of permissible values is a codomain. For example the integers may be defined as a codomain. All data values in an ADAMS database must belong

2. This project was developed using ADAMS, version 2. In places the syntax will differ from the more recent version 3; but other than minor syntax there are no significant differences between the versions.

to some defined codomain. An ADAMS codomain is analogous to a domain in the relational model.

Attribute: An attribute is a function taking an element to a codomain. It may be viewed as a field in a traditional database, and is used in this manner in the model for translating relational databases to ADAMS. All data values are obtained by applying attributes to ADAMS elements.

Map: A map is also a function, however it maps elements to other elements. Although obviously powerful, this construct is not needed to implement a relational database in ADAMS; and indeed, is regarded as an anathema in relational orthodoxy. It can be used to provide links such as are used in the network database model.

Class: A class is a type definition. It is a template describing the properties which instances of this type have. All ADAMS elements must belong to a class; they are instantiated members of the class. Typically a class definition lists the attributes and maps which may be applied to the class to obtain or store data values and other elements.

Set: Sets are the fundamental database structures in ADAMS.They are proper mathematical sets. The elements must all be of the same class and are unordered.

All of the above primitive constructs are given user defined names and are persistent (or not, if so desired). In addition there are pre-defined types corresponding to most of the primitive constructs, namely, 1) SET, 2) ATTRIBUTE and 3) MAP. These types may then be used to create other sets. For example a user can create a set of ATTRIBUTE elements. Such a set will just be an aggregation of various attributes as opposed to numerous instantiations of the same attribute. This raises an important point; all of the above constructs are definitions or templates from which ADAMS elements can be constructed. To create an ADAMS element, the user must create an instance of the class desired. This element may or may not be given a persistent name.

3.1 ADAMS Syntax

A brief description of the syntax of the language is provided here to ease the understanding of the translation system by providing some background into and samples of the actual ADAMS statements. Only the syntax needed to implement the relational database mode is discussed for more on the ADAMS language and its capabilities refer to [PFG89a]

3.1.1 Codomains

Codomains are defined and entered into the ADAMS dictionary by the codomain declaration statement. The definition currently specifies the form of legal values in the codomain and specifies the scope (discussed later in Section 3.2). A sample codomain declaration statement is:

```
CHAR isa CODOMAIN consisting of #[a-zA-Z 0-9$.-]+#.
```

This statement creates a codomain named *CHAR* with the values belonging to *CHAR* all consisting of one or more characters in a-z and A-Z or are digits or certain other characters. The codomain *CHAR* is persistent and available to other programs without redeclaration.

3.1.2 Attributes

There are several attribute statements which are pertinent to the discussion. First an attribute must be defined by an attribute declaration statement. As previously stated an attribute is a function from elements to codomain values. This statement assigns an image codomain to the attribute. Thus any values returned from the attribute will belong the codomain designated in the declaration. A typical attribute declaration statement is:

```
CHAR_ATTR isa ATTRIBUTE, with image CHAR.
```

This statement is a definition only. It adds the name and definition of the attribute to the dictionary but no memory is allocated for its use. The instantiation statement actually creates an instance of the attribute which can then be used to store and retrieve data A typical attribute instantiation looks like:

```
surname instantiates_a CHAR_ATTR.
```

It is often necessary to assign database values to host language variables or copy values from one ADAMS element to another. This is accomplished through the use of two assignment statements. The fetch statement is used to retrieve values from persistent storage into host language variables. A typical fetch statement is:

```
fetch into host_name from chairman.surname
```

where *host_name* is a variable of the host language, and *chairman.surname* is of the form <tuple.attribute> or more generally <element.attribute>. This is an application of the instantiated attribute *surname* to the instantiated element *chairman* and will return a value of the type of the image codomain for the attribute.

Another assignment statement used in the ADAMS code generated in the translation from SQL is the attribute assignment statement. This assigns the value from one <element.attrubute> to another. The syntax is that of a conventional assignment statement:

```
chairman.surname = assist_chairman.surname.
```

3.1.3 Classes

The class declaration provides a description of the properties of objects of the class. In particular, the declaration lists the attributes, maps, sets and other classes which are to be associated with the class. These associations are made by the **having** construct. Consider the following ADAMS statements.

```
first_name instantiates_a CHAR_ATTR
PERSON isa CLASS, having {first_name, surname}
chairman instantiates_a PERSON
```

These statements, including the preceeding *surname* instantiation and *CHAR_ATTR* declaration create an ADAMS element called *chairman*. This element has two attributes which may be applied to the element in order to store or retrieve data. As seen before in the attribute section, the attributes may be applied to the element with the period operator, e.g. *chairman.first_name*, to access the storage associated with the attribute and element

Classes may also be sets. This is the manner in which classes are used in the SQL to ADAMS translator. An example will be shown after sets are presented.

Finally, inheritance is supported among classes. That is, a class may be the type of another class and inherit the various attributes, maps, etc. that the parent class has. The parent attributes could then be applied to objects of the child class just as if those attributes were declared in the child classes declaration. This feature is not required for the implementation of a relational database but can make the creation of other databases more natural.

3.1.4 Sets

An ADAMS set is a set of instantiated elements belonging to the same class. This is fundamental ADAMS structure and is the primary construct in the relational emulation model.

As with attributes and classes, the use of sets requires both a declaration and an instantiation. A set is normally instantiated to be empty. Once instantiated, elements may be inserted or removed and set operations, such as union and intersection, may be performed on the set. Most important for mapping SQL to ADAMS is the ability to loop over all the members of a set using the **for_each** statement. This allows some action, any block of host and/or ADAMS statements, to be executed for every element in a set.

A set declaration specifies the type of elements belonging to sets of that class. An instantiation only names a particular set of that class. An example set declaration and instantiation are:

```
FACULTY isa SET of PERSON elements
cs_faculty instantiates_a FACULTY
```

A set may also have elements of the pre-defined types. The following example will occur again later when the relation definition in ADAMS is presented.

```
SCHEMA isa SET of ATTRIBUTE elements
```

An instantiated set may also be used in a class definition statement, as will be needed later. An example is:

```
SCHOOL isa CLASS, having {cs_faculty, ee_faculty}
```

This creates a class definition which has two sets *cs_faculty* and *ee_faculty* associated with it.

3.1.5 ADAMS Variables

ADAMS variables denote ADAMS elements. They are not declared to be of a particular type since the type can be determined by context. In fact, an adams variable can be used as any element type. An adams variable is declared by:

ADAMS_var variable_name

The variable may then be used to take on the value of an element. An example is the use of an adams variable in the **for_each** statement, as follows:

```
ADAMS_var professor
for_each professor in cs_faculty do
    some host and or ADAMS statements.
```

3.2 The ADAMS dictionary

Unlike the traditional database approach to persistent storage of database, file, record, and attribute, ADAMS provides a persistent name space. Names are used to identify specific elements just as if they were variables in a normal programming language. The ADAMS dictionary maintains information about the persistent names and provides a link between the names and the identifiers used for storage.

All codomains, attributes, maps, classes and sets are named. Actual instances may or may not be named as desired. All of the names in the examples provided so far would be placed in the dictionary had the example statements been actually executed.

Names are placed into the dictionary in a hierarchy. The level of an element is its scope and is set at declaration and instantiation. The levels of the hierarchy are:

SYSTEM
TASK
USER
LOCAL

The LOCAL level contains names which will only be used during the execution of a single program. These names will be removed from the dictionary at the termination of the program. Each user has a dictionary for their own private use. This is the USER level. Any name declaration without an explicit scope, such as in preceeding examples, are assumed to have USER scope by default. The TASK level provides names to be shared among users who form a group. And finally the SYSTEM level provides names to be shared among all users of the system. For example, the CHAR domain type is so universal that a declaration

```
CHAR isa CODOMAIN consisting of #[a-zA-Z0-9 -.$]+#, scope is SYSTEM
```

would define this term once and for all, so individual users need not repeat the definition. A more detailed presentation of the ADAMS dictionary may be found in [PFG89b].

In a persistent name space, particularly one with names of both types and instances, great care should be taken to create consistent and clear names. The name choices used in the next section reflect a desire to maintain clarity amidst a possibly confusing jumble of names.

4 The Model

The fundamental problem in mapping one database model onto another is the creation of corresponding data structures. As we have seen, ADAMS provides little in terms of the usual database structures. For example, the are no tuple constructs. In fact the only aggregation construct is the generic “set”. ADAMS does provide primitives which can be used to create data structures. This gives the flexibility to emulate other database models and is exactly the approach taken to map SQL to ADAMS.

In order to represent a relational database in the ADAMS language it is necessary to have ADAMS structures which correspond to relational domains, attributes, tuples and relations. Domains and attributes are simple data types and have corresponding ADAMS primitives. Relational schema, the resulting tuples and relations are complex data types which must be constructed. Fortunately both relational schema and relations are sets. ADAMS sets can be constructed which will represent the schema and relations. Tuples will be represented not by sets directly but by instances of a class which is associated with the set representing the schema. This allows the creation of multiple tuples as instances of the class.

Consider the SQL declaration for the *parts* relation in the familiar parts, suppliers, supplies database.

```
create table parts
    (
    p_nbr       integer,
    description char,
    price       real
    );
```

This relation will be used as an example to explain the translation. Domains and attributes are related and are building blocks for tuples so they will be studied first.

A relational domain is a set of data values. This has a clear correspondence with codomains in ADAMS and translates directly. A relational attribute contains a data value

belonging to the domain with which the attribute is associated. The ADAMS attribute is similar and so is used to represent the relational attribute. Now the ADAMS statements:

```
INTEGER      isa CODOMAIN consisting of #[-+ ][0-9]+#, scope is SYSTEM
INTEGER_ATTR isa ATTRIBUTE, with image INTEGER, scope is SYSTEM
p_nbr instantiates_a INTEGER_ATTR
```

represent the SQL fragment: *p_nbr integer.* SQL uses standard domains such as integer, real and character whereas ADAMS requires their declaration. It is expected that such "standard" codomains will be supplied at the SYSTEM dictionary level for any existing ADAMS systems.

Next, it is necessary to create ADAMS structures representing the relational schema and the relation itself. These are both just definitions. The tuples will just be instantiations of the schema and a relational instance will be an instance of the relation. As noted earlier a schema is just a set of attributes. Thus we will define a general schema to be:

```
SCHEMA isa SET of ATTRIBUTE elements, scope is SYSTEM
```

The SCHEMA definition is also expected to be widely used therefore the defintition is given SYSTEM scope. This definition uses the pre-defined class ATTRIBUTE to describe its elements. Any attribute instance may be inserted into a set which is of type SCHEMA. Such a definition gives a standard and very flexibility definition of a relational schema. To create a particular relational schema one only needs to instantiate the pertinent attributes, instantiate the schema and insert the attributes.

Consider the relational schema for the parts relation *parts(p_nbr, description, price)* where the domain of *p_nbr*, *description* and *price* are integer, char, and real respectively as in the definition. The ADAMS code to create this schema, assuming standard domains (codomains) is:

```
INTEGER_ATTR  isa ATTRIBUTE, with image INTEGER
CHAR_ATTR     isa ATTRIBUTE, with image CHARACTER
REAl_ATTR     isa ATTRIBUTE, with image REAL
p_nbr         instantiates_a INTEGER_ATTR
```

```
description   instantiates_a CHAR_ATTR
price         instantiates_a REAL_ATTR
SCHEMA        isa SET of ATTRIBUTE elements
parts_SCHEMA  instantiates_a SCHEMA
insert p_nbr       into parts_SCHEMA
insert description into parts_SCHEMA
insert price       into parts_SCHEMA
```

The code shown creates the relational schema from scratch. This needs only to be done once. Any codomain, attribute (definition or instance), or set which has already been entered into the dictionary is part of the persistent name space and can be used without additional declarations. Thus a second relational instance, say the supplies relation from our example, with the schema *supplies(s_nbr, p_nbr)* could be created after the previous example by just the following four statements:

```
s_nbr            instantiates_a INTEGER_ATTR
supplies_SCHEMA  instantiates_a SCHEMA
insert s_nbr into supplies_SCHEMA
insert p_nbr into supplies_SCHEMA
```

Next we can define a tuple using the schema definition and then define the relation and relation instance from the tuple definition. A class having the schema is declared. This gives a template from which tuples can be instantiated. The following statement illustrates a tuple definition from the previous *parts_SCHEMA*:

```
PARTS_TUPLE isa CLASS, having parts_SCHEMA
```

A relation is a set of tuples. Using the ADAMS set construct we can now create a relation and an instance of the relation by:

```
parts_RELATION isa SET of PARTS_TUPLE elements
parts instantiates_a parts_RELATION.
```

We have now created a relation instance from start to finish. The ADAMS code was interspersed with comments so the example is repeated to present the complete code to create a new relation in both SQL and in ADAMS. Again the SQL code for the **create table** statement for the *parts* relation is:

```
create table parts
    (
    p_nbr       integer,
    description char,
    price       real
    );
```

The code to create the equivalent relation in ADAMS, assuming the standard codomains exist, is:

```
INTEGER_ATTR    isa ATTRIBUTE, with image INTEGER
CHAR_ATTR       isa ATTRIBUTE, with image CHAR
REAL_ATTR       isa ATTRIBUTE, with image REAL
p_nbr        instantiates_a INTEGER_ATTR
description  instantiates_a CHAR_ATTR
price        instantiates_a REAL_ATTR
SCHEMA          isa SET of ATTRIBUTE elements
parts_SCHEMA instantiates_a SCHEMA
insert p_nbr       into parts_SCHEMA
insert description into parts_SCHEMA
insert unit_price  into parts_SCHEMA
PARTS_TUPLE     isa CLASS, having parts_SCHEMA
PARTS_RELATION  isa SET of PARTS_TUPLE elements
parts        instantiates_a PARTS_RELATION
```

Although the ADAMS code to create a relation is much longer than the SQL code, it should be pointed out that nearly half of the ADAMS code shown only needs to be executed once, then the names may be used freely. In addition, ADAMS is a mid-level language which only provides the primitives to allow users the flexibility to create a desired database, unlike SQL which operates solely in the relational model. Part of the purpose of the SQL to ADAMS translator is to permit the creation and use of relational databases by using a language dedicated to that model.

By viewing a relation as a set of tuples and a tuple as a set of attributes it is relatively straight forward to create relations using the ADAMS language. Relations are the fundamental structure in the relational model so it has now been shown that the fundamental structure in the relational model can be emulated in ADAMS.

Now that it has been demonstrated that relations can be emulated in ADAMS is remains to show that the relations created can actually be used in a reasonable fashion. The next two chapters detail the data manipulation and query portion of SQL to demonstrate that the relations created by the method described are actually useful.

5 The Translation of SQL's Data Manipulation Language

Once a relation has been created it is essential to be able to put tuples into the relation, remove tuples and find out what tuples exist in a relation. The SQL statements for update operations on relations are: insert, delete and modify. The dump statement allows inspection of all of the data in a relation. These data manipulation statements and their translations are discussed in this chapter.

5.1 The Insert Statement

Obviously it is necessary to put data into a database. In SQL this is accomplished with the insert statement. An insert statement specifies the table to be updated and the field names and values or just a list of values. An insert statement of the first type is:

```
insert into parts (p_nbr, description, price) values (101, bolt,.10 );
```

The translation into ADAMS is straight forward. A new row is to be inserted into the relation so a new row can be created as follows:

```
ADAMS_var row
row instantiates_a PARTS_TUPLE
```

First, an ADAMS variable is declared and then the variable is used to represent an element of type *PARTS_TUPL*E. This creates an unnamed ADAMS element, one that is not inserted into the dictionary. Since the row is of type *PARTS_TUPLE* it has a set containing the attributes in the relational schema. The data values listed are to be assigned to the appropriate attribute and then the new row should be inserted into the relation. This is accomplished by the following:

```
strcpy( host_var, “101” );
store from host_var into row.p_nbr
strcpy( host_var, “bolt” );
store from host_var into row.description
...
insert row into parts
```

Notice that host language statements are interspersed with ADAMS statements this is typical of ADAMS applications. The variable *host_var* is assumed to be a char array. The <element.attribute> notation, as in *row.p_nbr*, is used to access the actual data storage associated with the attributes.

The field names in the insert statement are optional. Without them the code generated would merely require a loop over the attributes in *parts_SCHEMA*. An example is not presented here but a similar situation arises in the dump statement and will be presented in that section.

5.2 The Delete Statement

Another important update function is the ability to remove tuples from relations. SQL's delete statement performs this duty.

The delete statement specifies the table name from which rows are to be removed and has a where clause specifying the condition for removing a row. Where clauses will be discussed in detail in Chapter 6 on the select statement. For now just note that deletion is made very flexible by this clause. A single row may be selected by equating all of the field values or selection may be far more complicated.

Deletion of rows is very similar to selection. The difference is of course the action taken when the appropriate rows have been identified. The ADAMS **remove** statement is employed to delete the selected rows.

5.3 The Update Statement

The update statement is a modification of existing rows. As with deletion, the update statement is primarily a select statement. The selected row the has the new values specified in the set clause stored into the appropriate attribute just as in the insert statement translation.

5.4 The Dump Statement

It may be necessary to view all of the data in a particular relation. The dump statement does precisely that, it dumps out all of the data values in a relation. The translation of this statement is of interest primarily because it demonstrates looping over the elements in the ADAMS sets and it suggests that the approach taken, that is to represent both tuples and relations as sets, was a good one.

The form of a dump statement is simple. It gives the name of the table to be dumped and an optional destination filename. For example:

```
dump table parts;
```

This simple statement requires two loops in the ADAMS implementation. One to obtain each row in the table and the other, a nested loop, to iterate over each field in the schema. The statement above translates into:

```
<<ADAMS_var row, attr                          >>
<<for_each row in parts do
<<    for_each attr in parts_SCHEMA do
<<          fetch into host_var from row.attr   >>
            printf( "%s", host_var );
            >>
      printf( "\n" );
      >>
```

This program fragment will dump all of the data values in the parts table.

In this ADAMS fragment we have included the ADAMS statement delimiters <<...>>, which we have previously omitted for clarity. Their primary purpose is to separate ADAMS statements from host language statements during preprocessing. But they also serve as loop end delimiters, as in the case of the last two >> delimiters. In following examples we will again elide these delimiters, and let indentation delimit the scope of ADAMS loops.

6 Translation of SQL's Select Statement

The query portion of SQL is the select statement. As the name implies selection is the central issue in the language. Unfortunately selection in SQL corresponds more closely to projection than to selection in the relational algebra, although it may be used to accomplish the relational algebra selection. That is, in general, the SQL selection statement retrieves fields rather than rows from the tables designated.

Three clauses comprise the select statement: the *select* clause, the *from* clause and the *where* clause. The *select* clause lists the fields that are to be selected. The *from* clause indicates all of the tables involved in the query. And the *where* clause, which is optional, specifies any condition on the retrieval of the fields in the select clause.

Examples of various select statements and their translation into ADAMS follow. The queries will be based on the Parts, Suppliers, Supplies database where these relations have the schema: Parts(p_nbr, description, price), Suppliers(s_name, s_nbr, city, zip) and Supplies(s_nbr, p_nbr).

6.1 Simple Select Statements

To demonstrate the basic select structure consider the query "Retrieve the descriptions of all parts that cost $1.00". In SQL the query is:

```
select description
from   parts
where  price = 1.00;.
```

This is a very simple query, only one table is involved and the *where* condition is also basic. The translation to ADAMS is also fairly simple.

The result of any select statement is a new relation. SQL implementations vary in their handling of the resulting relation. Some treat the relation as temporary and have a save statement, others have a *to* clause in the select statement to specify the file to which the resulting relation is saved. In the current implementation of the translator from SQL to ADAMS the result of a select statement is temporary. This is done by setting the scope of

the new relation to be LOCAL. To make the result persistent the user need only to add a rescope statement to the code generated.

Since the result of any select statement is a new table the first step in translation is the creation of a new table. This is very similar to the translation for the create statement except the field names are taken from the *select* clause. The relation created for the above query is:

```
result_SCHEMA   instantiates_a SCHEMA, scope is LOCAL
insert description into result_SCHEMA
result_TUPLE       isa CLASS, having result_SCHEMA, scope is LOCAL
result_RELATION    isa SET of result_TUPLE elements, scope is LOCAL
result          instantiates_a result_RELATION, scope is LOCAL.
```

In this translation, all select statements result in a relation with a well know name, namely *result*. So all queries will have generate a section of code similar to the code above with, of course, different attributes inserted into the schema. Note also that we have made all declarations LOCAL in scope. Like SQL, results are not persisten and vanish with the termination of the process.

Next the query itself must be translated. SQL is a descriptive language and must be translated into procedural code. To process the query, a loop is set up to inspect each row of the table to see if the where condition is met. If so, the field specified in the *select* clause is copied from the row into the resulting table. The ADAMS code for the query is:

```
ADAMS_var row, new_row
char temp[50];
for_each row in parts do
    fetch into temp from row.price
    if( (strcmp( temp, “1.00” ) == 0 )
          {
          new_row instantiates_a result_TUPLE
          new_row.description = row.description
          insert new_row into result
          }
```

There are several things to note here. First all comparisons are done in the host language and all values are character arrays. This is purely a restriction of the version 2 implementation of the ADAMS language that was used for this project. In version 3, this would be coded:

```
ADAMS_var row, new_row
for_each row in { x in parts | x.price = '1.00' } do
    new_row instantiates_a result_TUPLE
    new_row.description<-row.description
    insert new_row into result
```

This yields a much better implementation; however, the current implementation is sufficient to emulate SQL with the method shown.

A second, and slightly more complex example will use two tables and a conjunctive *where* clause. Retrieve the suppliers_ids, *s_nbr*, and part's *description* for all suppliers who supply parts that cost $1. In SQL this is:

```
select supplies.s_nbr, parts.description
from   supplies, parts
where  supplies.p_nbr = parts.p_nbr AND parts.price = $1;
```

SQL does not require the table names for the fields in the select clause unless it is ambiguous. For convenience of translation, table names are required by the current SQL to ADAMS translator, if there are two or more table used in the query.

Translation of this query is very similar to the previous example. A new result relation is created just as before, except the two attributes *s_nbr* and *description* are inserted into its schema. Then the query is processed. Since two relations are involved two loops are set up to provide all possible combinations of rows from each relation. This is actually a cartesian product. The testing is done as the combinations occur rather than after the cartesian product has been formed to collapse the steps. Once again if the test conditions are met the *s_nbr* and *description* fields are copied to the result table. The ADAMS code for the query is:

```
ADAMS_var new_row, row0, row1
char temp0[50], temp1[50], temp2[50]
for_each row0 in supplies do
    for_each row1 in parts do
          fetch into temp0 from row0.p_nbr
          fetch into temp1 from row1.p_nbr
          fetch into temp2 from row1.price
          if( (strcmp( temp0, temp1) == 0 )
           && (strcmp( temp2, "1.00" ) == 0) )
                {
                new_row instantiates_a result_TUPLE
                new_row.s_nbr = row0.s_nbr
                new_row.description = row1.description
                insert new_row into result
                }
```

This technique of creating the cartesian product and testing the condition as it is created is used for any number of relations involved in the query. Obviously this is not an optimal or even efficient method, but it does work. The goal of this project was to produce a working emulation of a relational database, not to focus on query optimization.

We should also note that this query is, in relational algebra terms, a join followed by a projection. The SQL condition `supplies.p_nbr = parts.p_nbr` will create the natural join of the two relations. The projection is then just the condition `part.price = 1.00`.

6.2 Selects Retrieving All Fields or All Rows

SQL supports selection of all fields from a table or tables and all rows from a table or tables. Selection of all fields, or equivalently selection of rows, is specified by using '*' in the select clause, *select* *. This indicates that all fields in the tables listed in the from clause are to be selected. This is easily accomplished in translation. First the result schema is created by inserting all the attributes in the tables listed in the from clause and then once a row has met the *where* condition a new loop is added which iterates over all the attributes in the result schema, assigning the values to the attributes of the row which will be inserted into the result. There is a restriction not present in SQL. That is, two tables having

fields with the same name can be duplicated in SQL; however in ADAMS the same name indicates a single attribute which can give only a single value when applied to an element[3]. This restriction can be overcome by creating a new attribute for one of the duplicates.

To select all rows from a table or tables the where statement is eliminated. This is equivalent to where **true**. And in fact this is the exact translation into ADAMS. That is, the if statement is still generated but the selection expression is just a ‘1’(i.e. if(1)).

6.3 Nested Queries

Two types of nested queries in SQL have been implemented in the translator. The first is the **in** condition and the second is the **exists** condition. Both of these create a new “sub relation” which is then used to test for inclusion or to test for emptiness respectively.

6.3.1 The In Function

The select **in** permits the use of the same attribute in different manners. For instance as the field to be selected and a field in the condition. An example will help to clarify the situation.

Retrieve the id numbers of all suppliers who sell the same parts as Jack’s store.

```
select s_nbr
from   supplies
where  p_nbr in (select  supplies.p_nbr
                 from    supplies, suppliers
                 where   supplies.p_nbr = suppliers.p_nbr AND
                         suppliers.s_name = ‘Jacks’);
```

In this query the field *s_nbr* is used both as the field to be retrieved and a field to restrict the rows returned. This provides more flexible and more powerful queries.

The translation of the **in** construct reflects the nested structure of the query. The query is translated from the inside out. First the nested sub-query is translated and creates

3. It is worth noting, that in this respect, the ADAMS interpretation is closer to the original relational model proposed by Codd than is the SQL interpretation. This is a true natural join.

a temporary relation. The outside sub-query then uses to the rows in the temporary relation to check against the rows in the relations in its from list to test if the field is the same. Once again, an example should clarify matters. The nested sub-query translates as the previous examples except the resulting relation is called *result1*. Ignoring the creation of the table *result*, which is as in all the previous examples, the translation for the outside sub-query is:

```
ADAMS_var row0, row_in, new_row
...
for_each row0 in supplies do
    for_each row_in in result1 do
          fetch into temp0 from row0.p_nbr
          fetch into temp1 from row_in.p_nbr
          if( (strcmp( temp0, temp1 )) == 0 )
                {
                new_row instantiates_a result_TUPLE
                new_row.s_nbr = row0.s_nbr
                insert new_row into result
                }
```

As clearly reflected in the translation, the **in** condition is an implicit equality test for the field in the *where* clause of the outer sub-query and the field selected in the inner sub-query. It is more though. It separates the two queries so that the *s_nbr* field can be used twice in the same query. In SQL this can also be accomplished by using aliases. Aliases provide a mechanism for using two copies of tables and so their same fields can be used in different ways. This can be accomplished in ADAMS by creating a new relation which is just a copy of the relation which has aliases.

6.3.2 The Exists Function

The **exists** expression is the set existential quantifier. In SQL **exists** checks the result of a nested query to see if any rows where selected, if so then the *where* clause is equivalent to where true and the selection occurs, otherwise no fields are selected. The test occurs for each row in the table in the *from* clause. The **not exists** expression can be used as the universal quantifier which is not actually present in SQL.

The translation of a query using **exists** is similar to the translation of a query using **in**. However, the nested sub-query must be evaluated for every iteration of the outer sub-query test. Thus the sub-query will actually be nested in the translation and not just occur once before the outer sub-query. Unlike the **in** function the temporary relation need not be created. It is sufficient merely to set a flag if the nested query evaluates true. This flag can then be tested.

As an example consider the english question: retrieve the supplier ids for all suppliers who sell at least one part which cost $1. The phrase “at least” indicates that this is an existence query. The SQL query is:

```
select s_nbr
from   supplies
where  exists   (  select parts.p_nbr
                   from parts, supplies
                   where parts.price = 1.00 );
```

The translation will have the normal creation of the *result* table, then the loop over all the rows in the supplies table. Nested in this loop will be the sub-query which selects the rows which have prices that are $1 and the test for the existence of any of them. The translation after the creation of the *result* table is:

```
ADAMS_var row0, row1, new_row
for_each row0 in supplies do
    exists = 0;
    for_each row1 in parts do
          fetch into temp0 from row0.p_nbr
          fetch into temp1 from row1.p_nbr
          fetch into temp2 from row2.price
          if( ( strcmp( temp0, temp1 ) == 0 ) &&
                 (strcmp( temp2, “1.00” )) == 0 ) )
                exists++;
    if( exists )
          {
          new_row instantiates_a result_TUPLE
          new_row.s_nbr = row0.s_nbr
          insert new_row into result
```

```
}
```

As with the **in** function, **exists** has a tacit equality test which is made explicit in translation, namely the comparison between the attribute *parts.p_nbr* and *supplies.p_nbr.* This is the connective between the sub-queries. The **not exists** qualifier is simply implemented by negating the test of exists.

6.4 Differences Between Tables and Sets

It was stated earlier that a relation was a set. This is the fundamental axiom of relational theory, leading to such results as “every relational schema is a superkey of the relation” [Mai83]. But set property of strictly distinct elements is seldom enforced in practical implementations. An SQL relation may contain more than one of the same tuple. This often occurs as a result of a query when two tuples are selected but the attribute which makes the tuples distinct is not selected. Usually duplicate tuples are not desired. SQL has a qualifier which may be used in the select statement to remove tuples that are identical. The keyword **distinct** may be inserted just after the keyword **select** in the statement to cause the removal of identical tuples. In some systems **distinct** is replaced with **unique**. To provide clarification there is an additional qualifier, **all**, to indicate that duplicate tuples will not be removed

Since removal of duplicate tuples is an expensive task in terms of computation time, it is typically done only if the qualifier is used. In the ADAMS translation the **distinct** qualifier is just a filter at the end of the query. Since the translation is obvious and is not of particular interest, no examples are provided.

6.5 Select Options Not Implemented

There are a number of options which SQL provides for use with the select statement that have not been implemented in the SQL to ADAMS translator. The major options not implemented are aggregation, ordering and explicit nulls.

Aggregation, for example: count, minimum, maximum, and average functions exist in SQL. These functions have not been implemented in the translator since they are just

processing the data. Clearly once the data has been retrieved, functions to operate on the data can be easily be implemented. That is a benefit of used a data management language embedded in a general purpose host language.

SQL is used as a commercial language and therefore has functions which are primarily for convenience. One of these is the *order by* function. This function sorts the tuples which result from a query by some field. Clearly this also could easily be accomplished once the data is obtained, by a host language filter.

In some instances it may be desirable to introduce null values into a database. This may be intentional or be the result of an update through a view. SQL allows the optional use of explicit null values. Each null value is distinct from the other but all belong the symbol NULL. It is felt that for our limited purpose null values are not necessary and have not been included in the current translator.

There are a variety of options, for example comparison functions which have not been implemented, many of which are system dependent. However, enough options have been implemented to fulfill our goal of building a working relational database in ADAMS.

7 Conclusion

ADAMS provides a small but flexible language for database management. It has been claimed that the language is powerful enough to support the three major database models. To support this claim a project was initiated to investigate the implementation of the relational model in ADAMS.

The goal of this project was to show that the relational database model could be emulated in ADAMS. To this end a lex-yacc based translator was constructed to convert the relational database language SQL into the ADAMS language. The language translation that occurs has been described in detail in this document. Although not every bell and whistle of SQL has been implemented in a corresponding ADAMS program, the core language has been successfully translated. Using the translation scheme described a working relational database can be, and in fact has been implemented in ADAMS. Thus demonstrating that the ADAMS language is sufficiently powerful to support the relational model.

8 Refrences

[Cod70] E. F. Codd, A Relational Model of Data for Large Shared Data Banks, *Comm. of the ACM 13, 6* (June 1970), 377-387.

[Mai83] D. Maier, *The Theory of Relational Databases*, Computer Science Press, Rockville, MD, 1983.

[PFG89a] J. L. Pfaltz, J. C. French, A. Grimshaw, S. H. Son, P. Baron, S. Janet, A. Kim, C. Klump, Y. Lin, L. Loyd, The ADAMS Database Language, IPC TR-89-002, Institute for Parallel Computation,Univ.of Virginia, Feb. 1989.

[PFG89b] J. L. Pfaltz, J. C. French, A. Grimshaw, S. H. Son, P. Baron, S. Janet, Y. Lin, L. Loyd, R. McElrath, Implementation of the ADAMS Database System, IPC TR-89-010, Institute for Parallel Computation,Univ.of Virginia, Feb. 1989.

[Weg87] P. Wegner, Dimensions of Object-Based Language Design, *Proc. OOPSLA '87*, Oct. 1987, 168-182.

Appendix A.

BNF Grammar for SQL Syntax as Implemented

<SQL_stmt> ::= <stmts>;

<stmts> ::= <stmt>
| <stmts> ; <stmt.

<stmt> ::= <create__stmt>
| <delete_stmt>
| <dump_stmt>
| <insert_stmt>
| <select_stmt>

<create_stmt> ::= **create table** <table_name> (<field_list>)

<field_list> ::= <field_domain>
| <field_list> , <field_domain>

<field_domain> ::= <field> <domain>

<delete_stmt> ::= **delete from** <table_name> <where_clause>

<dump_stmt> ::= **dump table** <table_name>
| **dump table** <table_name> **to** <file_name>

<insert_stmt> ::= **insert into** <table_name> (<field_list>) <i_cond>
| **insert into** <table_name> <i_cond>

<i_cond> ::= **from** <file_name>
| **values** (<const_list>)

<select_stmt> ::= <select_clause> <from_clause> <where_clause>

<select_clause> ::= **select** <distinct> <select_list>
| **select** <distinct> ‘*’

<distinct> ::=
| **distinct**
| **unique**
| **all**

<select_list> ::= <select_element>
| <select_list> , <select_element>

<select_element> ::= <path>

<from_clause> ::= **from** <table_name>
| **from** <from_list> , <table_name>

<where_clause> ::=
| **where** <predicate>

<predicate> ::= <condition>
| <condition> **and** <predicate>
| <condition> **or** <predicate>
| (<predicate>)

<condition> ::= <expr>
| <expr> **in** (<select_stmt>)
| **exists** (<select_stmt>)

<expr> ::= <expr> != <expr>
| <expr> == <expr>
| (<expr>)
| <const>
| <path>

<path> ::= <field>
| <table_name> . <field>

<const_list> ::= <const>
| <const_list> , <const>

<table_name> ::= **identifier**

<field> ::= **identifier**

<domain> ::= **identifier**

<file_name> ::= **identifier**

<const> ::= **number**

Appendix B.

YACC version of the SQL grammar implemented

The following grammar is the grammar which is used by yacc to develop the parser for the translator. There are some differences between the BNF grammar and the yacc grammar. This is due to parsing difficulties and to allow the easy expansion of the grammar to include more options.

```
sql:    {
        SetUpDictionary();
        OutputInit();
        }
    stats ';'
        {
        OutputEnd();
        ShutDownAdams();
        }
    ;

stats:
        { nest_level = 0; } statement
    | stats ';'
        { nest_level = 0; } statement
    ;

statement:
     create
    | delete
    | dump
    | insert
    | select
    ;

/*
Data Manipulation
*/

delete:
    DELETE FROM rec_alias where_clause
        {
        printAvars( $3 );
        delete_stmt( $3, $4 );
        }
    ;

insert:
    INSERT INTO table_name {strcpy(t_name,$3);} '(' field_list ')' icond
```

```
            {
            printf("<<\tADAMS_var row\t>>\n");
            printf("<<\trow instantiates_a %s_TUPLE\t>>\n\n",t_name);
            if( !((const_token *)$8)->from_file )
                constList_to_insertValList($8);
            list_insert_to_store($6,$8);
            printf("\n<<\tinsert row into %s\t>>\n",t_name);
            }
      | INSERT INTO table_name {strcpy(t_name,$3);} icond
            {
            if( !((const_token *)$5)->from_file )
                {
                printf("\tconst_token\t*const_list,\n\t*const_ptr;\n\n");
                constList_to_insertValLinkedList($5);
                }
            printf("<<\tADAMS_var row, attr\t>>\n");
            printf("<<\trow instantiates_a %s_TUPLE\t>>\n\n",t_name);
            loop_insert_to_store(t_name, $5);
            printf("\n<<\tinsert row into %s\t>>\n",t_name);
            }
      ;


rec_alias:
      path
      | path alias
      ;

icond:
      FROM filename
      {
      $$ = (char *)insertVal_fromFile($2);
      }
      | VALUES '(' const_list ')'
            {
            $$ = $3;
            }
      ;

/*
Query Langauge
*/

select:
      select_expr into_clause
      ;

select_expr:
      select_statement
      | select_expr UNION any select_statement
      | select_expr MINUS select_statement
      | select_expr DIVIDEBY select_statement
```

```
    | select_expr INTERSECT select_statement
    | '(' select_expr ')'
    ;

into_clause:
    | INTO filename
    ;

select_statement:
    select_clause FROM from_item_list select_options
        {
        if( nest_level > 1 )
            sprintf( table, "result%d", nest_level-1 );
        else
            strcpy( table, "result" );
        create_table( table, $1, $3 );
        printAvars( $3 );
        select_stmt( table, $1, $3, $4 );
        if( distinct_flag )
            {
            distinct( table );
            }
        nest_level--;
        $$ = expr_create( table );
        }
    ;

select_clause:
    SELECT unique sellist
        {
        $$ = $3;
        nest_level++;
        }
    | SELECT unique '*'
        {
        $$ = NULL;
        nest_level++;
        }
    ;

from_item_list:
    from_item
        {
        if( !exists($1) )
            {
            strcpy(err_str, ((list_item *)$1)->name );
            strcat(err_str, " not found in dictionary" );
            yyerror( err_str );
            }
        $$ = $1;
        }
    | from_item_list ',' from_item
```

```
            {
            if( !exists($3) )
                {
                strcpy(err_str, ((list_item *)$3)->name );
                strcat(err_str, " not found in dictionary" );
                yyerror( err_str );
                }
            $$ = (char *)append_list_item( $1, $3 );
            }
        ;

from_item:
        rec_alias
        ;

select_options:
            { $$ = NULL; }
        | WHERE predicate
            {
            $$ = $2;
            }
        ;

unique:
        | ALL           { distinct_flag = 0; }
        | DISTINCT { distinct_flag = 1; }
        | UNIQUE    { distinct_flag = 1; }
        ;

sellist:
        selelement
            {
            if( !exists($1) )
                 {
                strcpy(err_str, ((list_item *)$1)->name );
                strcat(err_str, " not found in dictionary" );
                yyerror( err_str );
                }
             }
        | sellist ',' selelement
            {
            if( !exists($3) )
                {
                strcpy(err_str, ((list_item *)$1)->name );
                strcat(err_str, " not found in dictionary" );
                yyerror( err_str );
                }
            $$ = (char *)append_list_item( $1, $3 );
            }
        ;

selelement:
```

```
        path
        ;

predicate:
        condition
        | condition AND predicate
                {
                $$ = exprs_merge( $1, " && ", $3 );
                }
        | condition OR predicate
                {
                $$ = exprs_merge( $1, " || ", $3 );
                }
        | '(' predicate ')'
                {
                $$ = $2;
                }
        ;

condition:
        expr
        | expr not IN in_sel_expr
                {
                if ( strcmp( $1, "not" ) == 0 )
                        change_item_info( $4, "in_not" );
                else
                        change_item_info( $4, "in" );
                $$ = $4;
                }
        | EXISTS '(' select_statement ')'
                {
                change_item_info( $3, "exists" );
                $$ = $3;
                }
        ;

not:
                {
                $$ = "";
                }
        |NOT
                {
                $$ = "not";
                }
        ;

any:
        |ALL
        |ANY
        ;

in_sel_expr:
```

```
        '(' select_statement ')'
                {
                $$ = $2;
                }
        ;

cond_sel_expr:
        expr
        | '(' select_statement ')'
        ;

const_recs:
        '(' const_list ')'
        | '[' const_rec_list ']'
        ;

const_rec_list:
        '(' const_list ')'
        | const_rec_list ',' '(' const_list ')'
        ;

path_list:
        path
        | path_list ',' path
        ;

field_list:
        field
        | field_list ',' field
                {
                append_attr_to_list($1,$3);
                }
        ;

path:
        path_e_list
        ;

path_e_list:
        path_element
                {
                $$ = (char *)create_list_item( strange);
                }
        | path_element
                {
                tmp = (char *)malloc( strlen( $1 ) + 1);
                strcpy( tmp, $1 );
                }
        '.' path_e_list
                {
                change_item_info( $4, tmp );
                $$ = $4;
```

```
            }
        ;

const_list:
        const
        | const_list ‘,’ const
            {
            append_to_const_list($1,$3);
            }
        ;

expr:
        expr LT any cond_sel_expr
            {
            $$ = exprs_merge( $1, “ < “, $4 );
            }
        | expr GT any cond_sel_expr
            {
            $$ = exprs_merge( $1, “ > “, $4 );
            }
        | expr EQ any cond_sel_expr
            {
            $$ = exprs_cmp( $1, “ == “, $4 );
            }
        | expr GE cond_sel_expr
            {
            $$ = exprs_merge( $1, “ >= “, $3 );
            }
        | expr LE cond_sel_expr
            {
            $$ = exprs_merge( $1, “ <= “, $3 );
            }
        | expr NE any cond_sel_expr
            {
            $$ = exprs_cmp( $1, “ != “, $4 );
            }
        | ‘(‘ expr ‘)’ /* parenthesis */
            {
            $$ = $2;
            }
        | const {
            $$ = expr_create( ((const_token *)$1)->val );
            }
        | path {
            add_rel( $1 );
            $$ = $1;
            }
        ;

const:
 STRING
            {
```

```
            $$ = (char *)create_const(yytext);
             }
      ;

expr_list:
      expr
      | expr_list ‘,’ expr
      ;

/*
Data Definition Language
*/
create:
      CREATE TABLE table_name {strcpy(t_name,$3);} tfd_list
            {
            create_table(t_name, $5);
            $$ = 0;
            }
      ;

tfd_list:
      field_dis {$$ = $1;}
      ;

field_dis:
      ‘(‘ tfd_fields ‘)’
            {
            $$ = $2;
            }
      ;

tfd_fields:
      tfd
            {
            $$ = $1;
            }
      | tfd_fields ‘,’ tfd
            {
            append_attr_to_list($1,$3);
            }
      ;

tfd:
      field o_domain_name
            {
            a_field_create($1,$2);
            }
      ;

o_domain_name:
 domain_name
       ;
```

```
type_name:
	IDENTIFIER
	;

dump:
	DUMP TABLE table_name {strcpy(t_name, $3);} to_clause
		{
		dump_table(t_name,$5);
		}
	;

to_clause:
		{
		$$ = NULL;
		}
	| TO filename
		{
		$$ = $2;
		}
	;

rfrom_clause:
	| FROM filename
	;

where_clause:
		{
		$$ = NULL;
		}
	| WHERE predicate
		{
		$$ = $2;
		}
	;

from_clause:
	| FROM path_list
	;

/*
IDENTIFIERS
*/
alias: IDENTIFIER ;
domain_name: IDENTIFIER ;
field: IDENTIFIER
	{
	 $$ = (char *)lookup_create($1);
	};
filename: IDENTIFIER ;
path_element: IDENTIFIER
	{ strange = (char *)malloc( strlen( $1 ) + 1 );
```

```
        strcpy( strange, $1 );
        }
        ;
record: IDENTIFIER;
table_name: IDENTIFIER ;
```