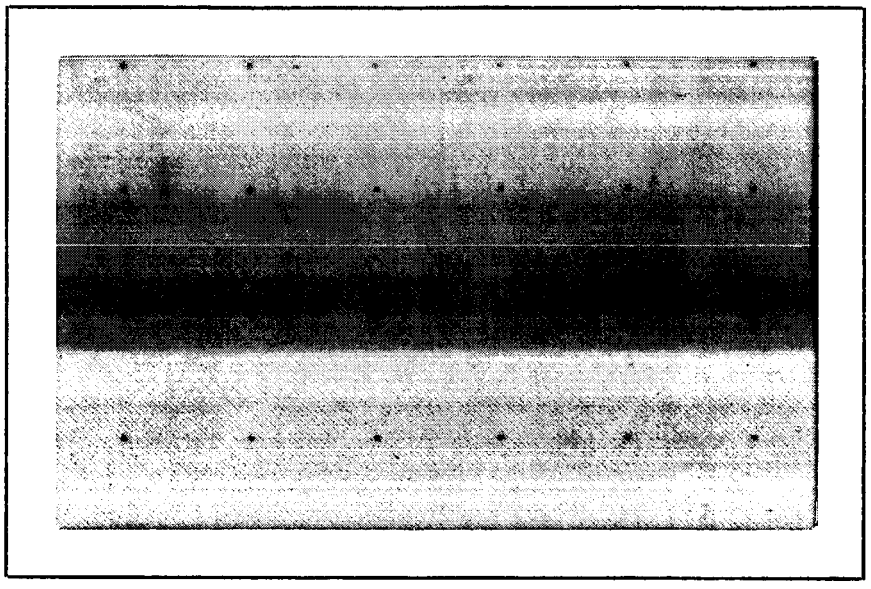


UNIVERSITY OF HOUSTON

DEPARTMENT OF COMPUTER SCIENCE

Houston, Texas 77004

(NASA-CR-180120) DATABASE INTERFACES ON NASA'S HETEROGENEOUS DISTRIBUTED DATABASE SYSTEM Semiannual Report, 11 Feb. 1987 (Houston Univ.) 35 p CSCL 05B G3/82

N87-16656

Unclas 43372

Semi Annual Report to
National Aeronautics and Space Administration

on

Database Interfaces on NASA's Heterogeneous Distributed Database System

Shou-Hsuan Stephen Huang
Department of Computer Science
University of Houston
Houston, Texas 77004

Feburary 11, 1987 Grant NAG 5-739

Contents

Semi Annual Report

1. Summary of Current Research

The purpose of DAVID interface module (Module 9: Resident Primitive Processing Package) is to provide data transfer between local DAVID systems and resident DBMSs. We shall summarize the result of current research here. A detailed description of the interface module can be found in Appendix 1.

In order to transfer database from resident DBMS to DAVID (or vice versa), we need to generate programs (called **database access programs** here) that can access both DBMSs. The purpose of the **Resident Processor** (Module 9.2) is to generate these database access programs. These database access programs can be written in (1) a host programming language with embedded database access statements for both database systems; or (2) a system command program with embedded data manipulation statements of the resident database system.

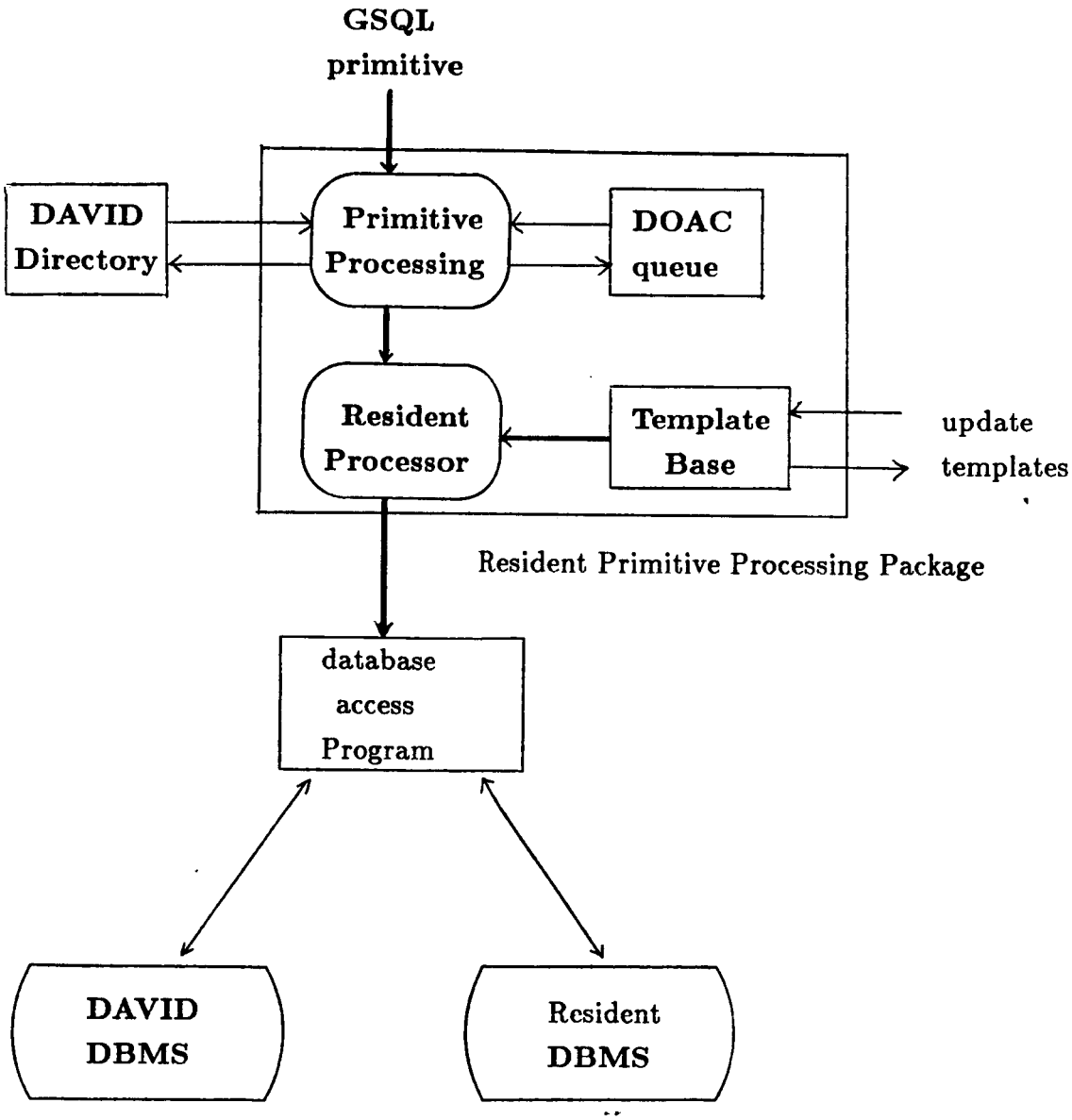
One of the most important objectives of the design of the interface module is to build the interface module for as many DBMSs as possible. To achieve the goal, we have to separate the knowledge about the resident DBMSs from the Resident Processor. So, our design of the interface module includes two major components: (1) a **Template Base** that contains a collection of templates (one for each type of primitives), and (2) a **Resident Processor** program that generates database access programs (one for each GSQL primitive).

Since the Resident Processor does not contain any knowledge about the resident DBMS, we need only one Resident Processor. This is a significant improvement from the initial design which calls for as many Resident Processors as the number of primitive types. The structure of interface module is illustrated in Figure 1 below.

In the last six months, we have constructed several Pascal templates which are included in Appendices 2–5. The Resident Processor program is also developed. Even though it is designed for the Pascal templates, it can be modified for templates in other languages, such as C, without much difficulties. The Resident Processor itself can be written in any programming language.

Since Module 5 routines are not ready yet, there is no way to test the interface module. However, simulation shows that the database access programs produced by the Resident Processor do work according to the specifications.

Figure 1: Interaction of Components of the Interface Module.



2. Proposed Research

A renewal proposal to continue the research on this project has been submitted to NASA. The project will be extended at no cost pending the approval of the grant application. Thus, we outline the goals for the next year below:

- completing the Template Base with additional primitives;
- converting the Pascal programs and templates into C;
- writing documentations and reports on the design and implementation;
- integrating the interface module with the routines Module 5 of the DAVID System;
- analyzing the performance of the interface module;
- developing interfaces with other DBMSs such as RIM.

Some of these tasks will be performed in conjunction with the researchers at SAR.

We have produce a technical report about the general design of the interface module. Two graduate students wrote their master theses on the topic of database interface. One of them is continuing her work at SAR.

Appendices

Appendix 1

Database Interfaces in Heterogeneous Database Management Systems

Shou-Hsuan Stephen Huang[1]
and
I-Hui (Irene) Lee[2]

Technical Report #UH-CS-87-1 January 1987

Department of Computer Science
University of Houston
Houston, Texas 77004

---

[1] This research is supported by a NASA Grant NAG 5-739.

[2] Current address for I. Lee: Science Application Research, 4400 Forbes Blvd., Lanham, MD 20706.

Abstract

This paper addresses the problem of database interface between a DBMS and the DAVID DBMS. DAVID (Distributed Access View Integrated Database) system is a distributed heterogeneous DBMS under development by National Aeronautics and Space Administration (NASA). Each site of the distributed system has its own local database system(s) together with a common DAVID system. Database can be moved from one local database system to another via the DAVID system which is capable of storing databases of all three models (relational, hierarchical, and network). Interface occurs when a database is moved from a local system into DAVID or vice versa.

Oracle, a relational DBMS, is used as an example of the local DBMS in this research. We present two different ways to interface these two DBMS's. The first method (called *HLI Approach*) uses the Host Language Interface (HLI) facility of the two systems to retrieve and store data. The second method (called *Arbi Approach*) does not use HLI facility, thus works for arbitrary database systems including DBMS's for microcomputers. Data are retrieved into a temporary file through the SQL interface of Oracle. A second program analyzes the data and stores them into DAVID by calling DAVID routines.

In order to generate the interface module efficiently, we use a template for each type of the queries in both approaches. A template is either a Pascal program with special commands embedded in it or a system command program that activates Oracle UFI interface and other program modules. The design of the templates will be discussed in detail. The method developed here may be used for interface between other systems.

1. Introduction

A distributed database is a database that is not stored in its entirety at a single physical location, but rather is spread across a network of computers that are geographically dispersed and connected via a communication link. A distributed database system is a system in which a user can access distributed database stored at any site of the network [1, 2, 7].

A distributed database may be stored in many database systems at many sites. If the systems are **homogeneous**, i.e., each site is running that same Database Management System (DBMS), the software task is relatively simple. Data can flow from one DBMS to another DBMS without changing its model and format. If the systems are **heterogeneous** then some modifications must be done before data can be moved from one DBMS into another.

This paper addresses the problem of **database interface** [6], i.e., how to evaluate a query involving one (or, in the case of join, two) DBMS and store the result into another DBMS at the same site. For example, we may want to retrieve a table from a relational DBMS and store the result in a hierarchical DBMS.

In a heterogeneous distributed DBMS, we need an interface module for each pair of different DBMS's. If there are n different DBMS's in the distributed system, we need $O(n^2)$ such interface modules. One way to reduce the number of interface modules is to have a common DBMS on top of all local DBMS's involved in the system. Thus, we need only n interface modules, one for each different DBMS.

Under this assumption, the interface is between a particular DBMS and the common DBMS (both at the same site). To move a database from one local DBMS to another, it must be transferred via this common DBMS.

Throughout this paper, we shall use Oracle DBMS as the local DBMS to be interfaced with the common DBMS. We choose DAVID system as an example of the common DBMS. DAVID (Distributed Access View Integrated Database) System is a distributed heterogeneous DBMS under development by National Aeronautics and Space Administration (NASA) [3, 4, 5]. It is built on top of existing DBMS's. A brief description of the DAVID system is given below.

The DAVID system is capable of storing databases of all three data models– relational, hierarchical and network. The main data object in the DAVID system is called a **cluster**. A cluster is a collection of one or more **tables**; and a table is a collection of zero or more rows. However, the domain of the attributes is more general than that of a relational system. The attribute values can be simple (atomic) or another table instance. Thus, a link in hierarchical or network model can be implemented as a "sub-table" within a DAVID clusters.

The structure of the retrieval statement in the DAVID system is a generalization of that of SQL. It produces a set of (result) clusters from a set of (source) clusters. The SELECT clause specifies (defines) the clusters selected. The FROM clause specifies the cluster(s) to select from. The WHERE clause specifies the conditions for the selection and may be omitted. Notice that the clusters selected do not have to have the same structure as that of the underlying cluster instance.

For example, the following statements create a hierarchical cluster **deptstaff** from a relational cluster **registrar** which is physically stored in Oracle DBMS as an Oracle table.

```
CREATE ACTUAL CLUSTER deptstaff
SELECT   (dept, staff) as deptstaff
         (id, name)    as staff
FROM     registrar
WHERE    ....;
```

The CREATE ACTUAL CLUSTER clause specifies that the resultant cluster instance will be saved in DAVID DBMS with the given name. Without it, the cluster will only be displayed but not saved.

In Section 2, we shall describe how the interface module between Oracle and the DAVID system works with two different approaches. Section 3 shows the design of the templates used in the interface module. Conclusions and discussions are given in the last section.

2. Interface Between Oracle and DAVID

The interface module is activated when it receives a primitive query from the DAVID system requesting a database transaction that involves both DAVID and Oracle. The most common primitive is **selection-projection** which retrieves data from a cluster stored in the Oracle DBMS and stores the result into the DAVID DBMS. **Semijoin** primitive joins a cluster stored in the Oracle DBMS and a DAVID

cluster stored in DAVID into a new cluster stored in the DAVID DBMS. `Store-to-database` selects data from DAVID and stores them into Oracle.

Definitions may also be transfered from one system to another. `Install` primitive allows users to store the definitions of existing Oracle databases into DAVID. `Define` primitive defines an equivalent DAVID database in Oracle.

In addition, there are `Drop`, `Insert`, `Delete` and `Update` primitives. Unless otherwise stated, we shall use selection-projection as the primitive in the following discussion. That is, data is transfered from Oracle to DAVID.

Assume that both DBMS's provide their own Host Language Interfaces (HLI's) through certain programming languages. We can write a, say Pascal, program that calls Oracle routines to obtain data from Oracle and put them in host program variables. Then these data can be manipulated and stored into DAVID DBMS according to the new cluster definition. In our case, this is done through DAVID routines with proper parameters. This is called the **HLI Approach** [6]. In the HLI Approach, we have to use a program in a host language. The result is stored into the DBMS directly.

The second approach is called the **Arbi Approach** because this approach does not use the HLI routines. Thus, it works for arbitrary database systems including those on microcomputers. This method is divided into two phases.

Phase I: Log on to DBMS and use the interface provided by the DBMS (such as the User-Friendly Interface of Oracle) to retrieve data. Store the result of the retrieval

into a temporary file, say Temp.dat.

Phase II: Use a (say Pascal) program to open the temporary file temp.dat and store the data into DAVID by calling DAVID routines one record at a time according to the definition of the result cluster.

In the Arbi Approach, we are using a "program" which consists of (1) operating system commands to bring up the DBMS; (2) database statements (such as SQL) to access the database; and (3) more system commands to execute the Phase II program.

In both cases, we need to generate a program for each primitive query based on the information provided in the primitive and system directory. This is not trivial since even a simple Pascal program that calls Oracle routines is not short.

Instead of generating programs that are very similar whenever a particular type of primitive is received, we can group the common parts of the HLI program together. These program skeletons (called **templates**) depend on the primitive types but are independent of the actual primitive instances. When a particular primitive is to be evaluated, we first find the corresponding template and try to *fill* the template with information available at that time (such as the names of the clusters to be selected from).

Since the templates are independent of any particular instance of the primitives, they do not know a lot of information. For example, they do not know the number of tables in a cluster, or the number of attributes in a particular table.

These information are available only after the actual primitive is received. So there must be some way to specify certain constructs of the template based on the actual primitive. For example, the number of variables to be DEFINEd in Oracle depends on the SQL statement contained in the primitive. So the template has to provide the syntax for the DEFINE routine and specify that it be repeated as many times as the number of columns in the SQL statement.

The detail design of the templates will be discussed in the next section.

3. Template Design

In order to make a template general enough, we have to use some special commands in the template. We shall discuss three types of commands here. These commands begin with a special character '@' and followed by a command name.

[1] Substituting Commands: These commands substitute command names with information obtained from the primitive or system directory. For example, in order to log on to both DBMS's, there are certain information that have to be provided such as Oracle user-id and password. In the template, whenever we need such information, a command such as `@Ouserid` and `@Opassword` will be used.

The general syntax for a substituting command is:

```
@<name>[-<sequence number>]
```

where <name> is a string of letters. The sequence number is used when there are more than one instances to be substituted. For example, we may have to use table

names of a cluster which consists of three tables. They will be called **@Tablename-1**, **@Tablename-2**, and **@Tablename-3**. In fact, these sequence numbers are generated by using repeating commands discussed below.

[2] Repeating Commands: In many situations, a certain string or statements in a template have to be repeated many (but unspecified) times. The repeating commands identify the string and determine the number of time to duplicate the string.

The syntax of a repeating command is given below:

@begin<name>[(<separator>)]
string to be repeated
@end<name>

When a string is to be repeated n times, a separator will be used $n-1$ times to separate them. The separator can be one of the following: blank (default), ",", ";", or "OR" depending on the syntax.

If the <name> is "cluster", the string has to be repeated for the number of clusters involved in the FROM or SELECT clause. For **@begintable**, the string has to be repeated for as many times as there are tables in a given cluster. Most of the time, there is a three-level nesting of commands of the form cluster-table-column.

There are two numbers associated with a repeating group: an index and a sequence number. The **index** is an integer indicating the number of times the string has to be repeated. It is initialized to one whenever the repeating command

is encountered. (Since repeating commands may be nested, the same repeating group may be encountered more than once in a template.) The index is incremented whenever it corresponding @end command is reached.

The **sequence number**, on the other hand, is a cumulative index. It works like an index but does not reinitialize to one. So it represents the actual number of times a string has been repeated whereas the index is relative. If there is a substituting command (say, @xxx) in the string within a repeating group, then the command name will be appended by the sequence number of the repeating group. Thus the command name plus the sequence number (@xxx-*i*) is unique. The unique identification allows us to perform the substitution properly.

The index of a repeating group can also be used inside the repeating group. Detail of the indexing commands will be discussed in the next subsection.

[3] Indexing commands: The index of a repeating group may be accessed by @*i* where *i* indicates the repeating group by its level of nesting with @1 referring to the outermost repeating group.

With the three types of commands defined above, we can describe the process of converting a template into a program. The interface module consists of a program that *transforms* the template into a program. The transformation process is divided into two steps (logically): First, the repeating commands are evaluated. The indexes and sequence numbers are inserted into the template at this time. Then the substituting commands are replaced by their contents.

A simple example using Arbi Approach is given below. This template defines tables in Oracle given a cluster definition in DAVID. We assume the cluster consists of three tables S, P and SP with 3, 4 and 2 columns respectively. In this case, the transformation program gets information from the primitive about the cluster name. Then it gets the definition of the cluster from the DAVID system dictionary.

Template before Transformation:

```
$ufi
   @OUSERID/@OPWD
   @BEGINTABLE
      create table @TABLENAME (
         @BEGINCOLUMN(,)
            @COLUMN @CTYPE (@LENGTH)
         @ENDCOLUMN );
   @ENDTABLE
   exit
```

Template after Repeating:

```
$ufi
   @OUSERID/@OPWD
      create table @TABLENAME-1 (
            @COLUMN-1 @CTYPE-1 (@LENGTH-1),
            @COLUMN-2 @CTYPE-2 (@LENGTH-2),
            @COLUMN-3 @CTYPE-3 (@LENGTH-3)
      );
      create table @TABLENAME-2 (
            @COLUMN-4 @CTYPE-4 (@LENGTH-4),
            @COLUMN-5 @CTYPE-5 (@LENGTH-5),
            @COLUMN-6 @CTYPE-6 (@LENGTH-6),
            @COLUMN-7 @CTYPE-7 (@LENGTH-7)
      );
      create table @TABLENAME-3 (
            @COLUMN-8 @CTYPE-8 (@LENGTH-8),
            @COLUMN-9 @CTYPE-9 (@LENGTH-9)
```

```
        );
    exit
```

Template after Transformation:

```
$ufi
    cosc123/abcdefgh
        create table s (
                sname char (10),
                sno number (4),
                city char (10)
        );
        create table p (
                pno number (6),
                pname char (10),
                city char (10),
                color char (6)
        );
        create table sp (
                sno char (4),
                pno char (6)
        );
    exit
```

The next example shows how several tables are installed into DAVID (as a cluster) using suppliers-parts example. The spool command sends the output from the retrieval statement to a file for later processing. The file is assigned a unique file name based on the primitive. A second program, install.exe in this case, accesses the temporary file and then converts the definitions of the three tables into the definition of a cluster. Then the cluster definition is inserted into the DAVID system dictionary.

Template before Transformation:

```
$ufi
```

```
      @OUSERID/@OPWD
      spool @TEMPFILE
      select tname, cname, coltype, width
      from   col
      where
         @BEGININSTALL(OR)
            tname = '@INTABLE'
         @ENDINSTALL  ;
      exit
   $run install.exe
   @TEMPFILE
```

Template after Repeating:

```
   $ufi
      @OUSERID/@OPWD
      spool @TEMPFILE
      select tname, cname, coltype, width
      from   col
      where
            tname = '@INTABLE-1' OR
            tname = '@INTABLE-2' OR
            tname = '@INTABLE-3'   ;
      exit
   $run install.exe
   @TEMPFILE
```

Template after Transformation:

```
   $ufi
      cosc123/abcdefgh
      spool or12345
      select tname, cname, coltype, width
      from   col
      where
            tname = 'S' OR
            tname = 'P' OR
            tname = 'SP'   ;
      exit
```

```
$run install.exe
or12345
```

The third example uses HLI Approach. Since a template for HLI approach is very large in size, we can only show an abbreviated (and somewhat simplified) version of the template.

Selection-Projection Template:

```
......
Const
  OUID = '@OUSERID';
  OPWD = '@OPWD';
....
Var
  @BEGINSVAR
     s@SVAR: string20;
     r@SVAR: string20;
  @ENDSVAR
...
Begin
   ......;
   @BEGINSVAR
      odfinn (curs, @1, s@SVAR, 20, 1);
   @ENDSVAR
   ......;
   @BEGINCLUSTER
      gsql_run (vca, define@1, flag, result@1);
      ....;
      @BEGINTABLE
         tablename := '@TABLENAME';
         @BEGINCOLUMN
            COLUMN := '@COLUMN';
            bindcolumn (cva, cca, tablename, column,
                        r@SVAR, @TYPE, @LENGTH);
```

```
        ....;
      @ENDCOLUMN
    @ENDTABLE
    open (f@1, 'f@1.dat', new);
    rewrite (f@1);
  @ENDCLUSTER
  ......;
```

4. Conclusions

Two approaches to interface databases in a heterogeneous distributed system is introduced in this paper. Both methods use templates to build programs for the transformation of a database from one DBMS to another.

We summarize some of the important features of the interface design. (1) The commands introduced in this paper can be used in both types of templates–HLI and Arbi approaches. (2) The commands are independent of the host programming languages. We use Pascal as the host language here. To change to a different host language, the same commands can be used without much change. Some extensions, such as adding a different separator in repeating commands, may be necessary. (3) The commands can be used for any type of DBMS's. (4) For HLI approach, the program to transform the templates into database access programs is independent of the templates. We need only one program module for all HLI templates.

References

[1] C. J. Date: *An Introduction to Database Systems*, Vol. 1, 4th ed., Addison-Wesley, 1985.

[2] S. Ceri and G. Pelagatti: *Distributed Databases: Principles and Systems*, McGraw-Hill, 1984.

[3] B. Jacobs: *On Database Logic*, Journal of ACM, 29, 2, pp. 310-332, 1982.

[4] B. Jacobs: *Applied Database Logic, Vol I: Fundamental Database Issues*, Prentice-Hall, 1985.

[5] B. Jacobs: *Applied Database Logic, Vol II: Heterogenous Distributed Query Processing*, Prentice-Hall, to appear.

[6] I. H. Lee: *Database Interface Between Oracle and DAVID: A Host Language Interface Approach*, M.S. Thesis, The University of Houston–University Park, December 1986.

[7] J. Ullman: *Principles of Database Systems*, 2nd ed., Computer Science Press, 1982.

Appendix 2: Template of Define Cluster

```
$ufi
   @OUID/@OPWD
@BEGINTABLE
Create      .
TABLE @TABLENAME (@BEGINCOLUMN_C @COLUMN @TYPEC (@LENGTH) @ENDCOLUMN);
@ENDTABLE
exit
```

Appendix 3: Template of Drop Cluster

```
$ufi
 @OUID/@OPWD
 @BEGINTABLE drop table @TABLENAME;
 @ENDTABLE
 exit
```

Appendix 4: Template of Install Cluster

```
$ufi
   @OUID/@OPWD
   spool @TEMPFILE
   select tname, cname, coltype, width from col
   where @BEGININTABLE_0 tname = '@INTABLE' @ENDINTABLE ;
   exit
$run install.exe
@TEMPFILE
```

Appendix 5: Template of Selection Projection

```
Program SelMulProj (input, output);
CONST
   EndOfTable = 4;
   (* ORACLE userid and password *)
   UidPwd = '@OUID/@OPWD';
   (* DAVID uid and pwd *)
   UID = '@DUID';
   PWD = '@DPWD';
   (* DAVID result  name *)
   @BEGINCLUSTER RESULT@1 = '@CLUSTER'; @ENDCLUSTER
TYPE
   string5 = packed array [1..5] of char;
   string10 = packed array [1..10] of char;
   string20 = packed array [1..20] of char;
   string15  = packed array [1..15]  of char;
   string120 = packed array [1..120] of char;
   string400 = packed array [1..400] of char;
   Vstring100 = Varying [100] of char;
   vcaType = Record
                  code :  integer;
                  others :  string5;
             end;
   CcaType = string5;
   GcaType = string5;
   SccaType = string5;
   SourceType = Vstring100;
   ViewNameType = Vstring100;
   UidType = string10;
   PasswdType = string10;
   QueryType = string400;
   FlagType = string5;
   IdstringType = string5;
   TableNameType = string20;
   ColumnType = string20;
   ProgvarType = string20;
   ProgTypeType = integer;
   Ptr_Vca = ↑VcaType;
   Ptr_Gca = ↑GcaType;
   Ptr_Cca = ↑CcaType;
   ptr_scca = ↑Sccatype;
   word   = [word] -32768..32767;
   byte64 = array [1..32] of word;
VAR
   i, j, k :  integer;
```

```
   (* CURSOR Area and SQL Communications Area *)
   CURS, SQLDCA: byte64;
   (* DAVID Variables *)
   duid :  uidtype;
   dpwd :  passwdtype;
   vca :  VcaType;
   pvca :  Ptr_Vca;
   cca :  CcaType;
   pcca :  Ptr_Cca;
   gca :  GcaType;
   pgca :  Ptr_Gca;
   scca :  SccaType;
   source :  SourceType;
   ViewName :  ViewNameType;
   flag :  FlagType;
   idstring :  IdstringType;
   tablename :  TableNameType;
   column :  ColumnType;
   progvar :  ProgvarType;
   proglength :  integer;
   (* ORACLE query and Result Definition *)
   query :  QueryType;
   @BEGINCLUSTER define@1:  QueryType;
   C@1 :  ptr_cca;
   s@1 :  ptr_scca;
   f@1 :  TEXT; @ENDCLUSTER
   (* Source and Result Variables *)
   @BEGINSVAR s@SVAR : packed array [1..20] of char;
   r@SVAR : packed array [1..20] of char; @ENDSVAR

Procedure gsql_connect   (uid :  UidType;
                          passwd :  PasswdType;
                          pvca :  Ptr_Vca;
                          viewname :  ViewNameType); extern;
Procedure gsql_run       (vca :  VcaType;
                          query :  QueryType;
                          flag :  FlagType;
                          source :  sourcetype); extern;
Procedure gsql_compile   (vca :  VcaType;
                          pgca :  Ptr_Gca;
                          query :  QueryType;
                          flag :  FlagType;
                          idstring :  IdstringType); extern;
Procedure gsql_eval      (vca :  VcaType;
                          gca :  GcaType;
                          flag :  FlagType;
                          idstring :  IdstringType); extern;
Procedure asgcluster     (vca :  VcaType;
```

```
                          pcca :  Ptr_Cca;
                          source :  SourceType;
                          typ :  char); extern;
Procedure bindcolumn      (vca :  VcaType;
                          cca :  CcaType;
                          TableName :  TableNameType;
                          column :  ColumnType;
                          progvar :  ProgvarType;
                          progtype :  integer;
                          proglength :  integer); extern;
Procedure asgsubcluster   (vca :  VcaType;
                          cca :  CcaType;
                          scca :  SccaType;
                          source :  SourceType); extern;
Procedure scrinsert      (vca :  VcaType;
                          pcca :  ptr_cca;
                          scca :  ptr_Scca); extern;
Procedure deasgncluster (vca :  VcaType;
                          pcca :  Ptr_Cca); extern;
Procedure gsqldrop       (vca :  VcaType;
                          pgca :  Ptr_Gca); extern;
Procedure logoff         (uid :  UidType;
                          pvca :  Ptr_Vca;
                          viewname :  ViewNameType); extern;
Procedure rollback       (vca :  VcaType); extern;

(* ORACLE Procedures *)
Procedure Olon    (Var OSQLDCA : byte64;
                       OUId :  string15;
                       OUIdLen :  integer); EXTERN;
Procedure Oopen   (Var OCURS : byte64;
                   Var OSQLDCA : byte64); EXTERN;
Procedure Osql3   (Var OCURS : byte64;
                   Var Osqlstmt :  string400;
                       OsqlLen :  integer); EXTERN;
Procedure Odfinn (Var OCURS : byte64;
                       Oposition :  integer;
                   Var Obuffer :  string20;
                       Obufl :  integer;
                       Oftype :  integer); EXTERN;
Procedure Oexec   (Var OCURS : byte64); EXTERN;
Procedure Ofetch (Var OCURS : byte64); EXTERN;
Procedure Oclose (Var OCURS : byte64); EXTERN;
Procedure Ologof (Var OSQLDCA : byte64); EXTERN;
Procedure Oermsg (Var OCURS : integer;
                   Var Msgbuf :  string120); EXTERN;

Procedure ErrorO (m, n :  integer);
Var
```

```
   i :  integer;
   msg :  string120;
   err :  TEXT;
Begin
   for i := 1 to 120 do
      msg [i] := ' ';
   Oermsg (n, msg);
   open (err, 'erro.dat', new);
   rewrite (err);
   Writeln (err, 'Error occurred during ORACLE ');
   case m of
     1 :  writeln (err, 'OPEN ',msg);
     2 :  writeln (err, 'LOGON ', msg);
     3 :  writeln (err, 'SQL ', msg);
     4 :  writeln (err, 'DEFINE ', msg);
     5 :  writeln (err, 'FETCH ', msg);
     6 :  writeln (err, 'EXECUTE ', msg);
   end;
   close (err);
End;

Procedure ErrorD (n :  integer);
Var
   err :  TEXT;
Begin
   write ('Error during DAVID ');
   Case n of
      1 :  writeln (err, 'gsql conncet');
      2 :  writeln (err, 'gsql run');
      3 :  writeln (err, 'gsql compile');
      4 :  writeln (err, 'gsql evaluate');
      5 :  writeln (err, 'assign cluster');
      6 :  writeln (err, 'bind column');
      7 :  writeln (err, '');
      8 :  writeln (err, 'assign subcluster');
      9 :  writeln (err, '');
      10 :  writeln (err, 'assign subcluster row');
      11 :  writeln (err, 'deassign cluster');
      12 :  writeln (err, 'gsql drop');
      13 :  writeln (err, 'log off');
   end;
end;

Procedure Build_Query;
Var
   i, j, k :  integer;
   temp :  string20;
Begin
   k := 1;
```

```
   @BEGINOQUERY temp := '@OQUERY';
   for j := 1 to 20 do begin Query [k] := temp [j]; k := k + 1; end; @ENDOQUERY
   @BEGINCLUSTER
   k := 1;
   @BEGINDEF temp := '@DEF';
   for j := 1 to 20 do begin Define@1[k] := temp [j]; k := k + 1; end; @ENDDEF
   @ENDCLUSTER
   for i := 1 to 132 do write (query [i]); writeln;
End;

Procedure Convert (var f :  text; t :  string20);
var
   i, j, k, m, n, num :  integer;
   need :  boolean;
Begin
   i := 1; need := true;
   while (t [i] = ' ') do i := i + 1;
   while (i <= 20) and need do begin
      if not (t [i] in ['0'..'9']) then need := false;
      i := i + 1;
      end;
   i := 1; k := 0; num := 0;
   if need then begin
      while (t [i] = ' ') do i := i + 1;
      for j := 20 downto i do begin
         num := num + (ord (t [j]) - ord ('0')) * 10 ** k;
         k := k + 1;
         end;
      write (f, num :  6);
      end
   else begin
      m := 20; while (t [m] = ' ') do m := m - 1; write (f, '  ');
      if (m > 10) then for n := 1 to 20 do write (f, t [n])
      else for n := 1 to 10 do write (f, t [n]);
      end;
end;

Begin
  duid := pad (uid, ' ', 10);
  dpwd := pad (pwd, ' ', 10);
  Build_Query;
  Olon (SQLDCA, UidPwd, 15);
  If (CURS [1] <> 0) Then Error0 (1, CURS [1])
  Else Begin
    OOpen (CURS, SQLDCA);
    If (CURS [1] <> 0) Then Error0 (2, CURS [1])
    Else Begin
      Osql3 (CURS, Query, 400);
      If (CURS [1] <> 0) Then Error0 (3, CURS [1])
```

```
Else Begin
  @BEGINSVAR ODFINN (CURS, @1, s@SVAR, 20, 1); @ENDSVAR
  If (CURS [1] <>0) Then Error0 (4, CURS [1])
  Else Begin
    gsql_connect (duid, dpwd, pvca, viewname);
    if (vca.  code < 0) then errord (1)
    else begin
      @BEGINCLUSTER
      gsql_run (vca, DEFINE@1, flag, RESULT@1);
      if (vca.  code < 0) then errord (2)
      else begin
        asgcluster (vca, c@1, RESULT@1, 'W');
        if (vca.  code < 0) then errord (5)
        else begin @BEGINTABLE
        tablename := '@TABLENAME'; @BEGINCOLUMN  column := '@COLUMN';
        bindcolumn (vca, cca, tablename, column, r@RVAR,@TYPE, @LENGTH);
        if (vca.  code < 0) then errord (6) else begin @ENDOLUMN @ENDTABLE
        gsqldrop (vca, pgca);
        if (vca.  code < 0) then errord (12)
        else begin
        asgsubcluster (vca, cca, scca, RESULT@1);
        if (vca.  code < 0) then errord (8)
        else begin
        open (f@1, 'f@1.dat', new); rewrite (f@1); @ENDCLUSTER
          OExec (CURS);
          If (CURS [1] <> 0) Then Error0 (5, CURS [1])
          Else Begin
            Repeat
              OFetch (CURS);
              if (CURS [1] <> EndofTable) then begin
              @BEGINCLUSTER @BEGINTABLE @BEGINCOLUMN
              r@RVAR := s@RVAR; convert (f@1, s@RVAR);
              @ENDOLUMN @ENDTABLE writeln (f@1);
              scrinsert (vca, c@1, s@1);
              if (vca.  code < 0) then begin
                rollback (vca);
                errord (10);
                end;
                @ENDCLUSTER
               end;
             Until (CURS [1] <> 0) or (CURS [1] = EndOfTable)
                or (vca.  code < 0);
            end;
            @BEGINCLUSTER
            end;
            close (f@i); end;
            @BEGINTABLE @BEGINCOLUMN end; @ENDOLUMN @ENDTABLE
            end;
```

```
                deasgncluster (vca, pcca)
                if (vca.  code < 0) then errord (11);
                end;
                @ENDCLUSTER
              end;
            logoff (duid, pvca, viewname);
            if (vca.  code < 0) then errord (13);
            End;
        End;
      End;
    End;
  OClose (CURS);
  OLogof (SQLDCA);
End.
```