New ⇒ Data streams + Jim's comments

OPEN-SQL

Jim Gray
Don Slutz
Ramarao Yendluri

02/01/88

$dino.hetero.ospaper

Printed on 2/17/88

1.0 INTRODUCTION

The Relational Model of data was introduced by Dr. E.F. Codd in 1969. Since then many vendors have developed Database Management Systems (DBMS) based on the relational model. Some of the popular systems are ORACLE, INFORMIX, DB2, RTI, NonStop SQL and SYBASE. Most of these systems were developed as a single node database management systems. Some of them were enchanced to provide distributed access to the data. Some systems were designed for homogeneous distributed access.

Along with the Relational Model, the SQL language was developed in the IBM Research Labs in 1974. Within SQL, a user can create data, access data and/or update data within a database. SQL in some form was implemented by many of the relational DBMS vendors. Even though SQL was invented in 1974, it did not become a standard until October 1986. To date, the SQL DBMS vendors have not adopted the SQL language in its entirety. Some vendors implemented a subset, some vendors made extensions and some vendors altered the original SQL syntax to suit their environment. The SQL standard calls for certain Data Types but vendors have implemented the data types with some extensions. Thus, even though every vendor "supports SQL" they are not fully compatible.

To provide data integrity, every vendor provides a transaction protection mechanism in their database systems. But, there too they differred in actual implementation. Some vendors provided a distributed transaction capability and some did not.

Distributed databases are important for economical, organizational and technological reasons. Today, database systems are available on PCs, Minis, and Mainframe Computers. Users are demanding connectivity between the PC/workstation, Mini computer, and Mainframe based databases. In this environment, users would like to write applications on PCs and workstations which access data from the local database(s) as well as mainframes and minis. This kind of access is said to be heterogeneous database access.

A heterogeneous Distributed Database Management System uses at least two different DBMSs. The DBMSs may be running on the same hardware and same operating system or different hardware and different operating systems. The users are demanding access to heterogeneous database systems. For example, the users may have a network of PCs, and a single Corporate Mainframe or a network of main frames. The PCs are running applications accessing data in the local node, local network, and in the main frame nodes.

1.1 WHY OPEN SQL?

Even though most of the relational database vendors support SQL, there are SYNTAX differences, FUNCTIONAL differences (some support outer joins and some do not etc), SCHEMA differences (CREATE TABLE synatx is usually different), DATA TYPES differences (some support DECIMAL numbers, some do not; some support BINARY data and some call it RAW data; and many more), CATALOG differences (some call it SYSTABLES, some call it TABLES, and some call it SYSOBJECTS) in naming, and TRANSACTION management differences (BEGIN TRAN, BEGIN WORK, no support for BEGIN WORK, etc), and DATA STREAM differences.

With these differences it is difficult for a user to write a single application that can access more than one vendor's database. Without a standard the user could be faced with different user interfaces to different database systems, and the database vendors would find it difficult to supply PC/workstation software that exploited the different capabilities of each PC/workstation.

1.2 WHAT IS OPEN SQL?

OPEN SQL is an attempt to solve the above problems and define a standard to facilitate access to databases from workstations/PCs and other database systems. OPEN SQL defines a SQL syntax, Data Types, Catalogs, Transaction Model, Formats, and Protocols to fulfill users demands. This standard is defined in two parts, a) Applications Programming Interface (API) and b) Format And Protocols (FAP). The details of API and FAP are discussed in the subsequent sections.

1.3 ADVANTAGES OF OPEN SQL

If the OPEN SQL standard is adopted by most vendors, it allows applications to access data on different SQL systems using a standard SQL API. This allows the users and third-party vendors to write applications that are <u>portable</u> from machine to machine. It allows users to mix data on PCs, departmental systems, and mainframes. It also allows DB2, Tandem, Sybase, ORACLE and other vendors to coexist. In addition, OPEN SQL interface provides independence such that a user may use the <u>same front end</u> to access several different databases, and a single database may be shared by a population of users with different PCs and workstations.

This document is intended to highlight the importance of OPEN SQL, and define the OPEN SQL components (API and FAP).

2.0 WHAT OTHERS ARE DOING?

RTI, ORACLE, Teradata, and Gupta Technology all provide connectivity to their systems running on different hosts (heterogeneous hardware but homogeneous database systems). This connectivity uses their own protocols. However, ORACLE can not talk to RTI, or Teradata, or Gupta. Others also have the same problem.

ORACLE, GUPTA, RTI and Teradata have announced exchange of data with DB2 and IMS (using IBM's DXT). ORACLE allows access to DB2 data in a passthrough mode. This was acheived by writing servers on the IBM machines.

In addition, an ISO committee is defining a standard for Remote Data Access (RDA). The aim of the RDA standard is to define the interworking between two program components in different end systems, where one controls a database and the other requires to read and update data. In this, they are defining their own 2-phase commit protocols, server access protocols, data types of ISO and inherit ISO/ANSI defined SQL. The standard specifies that the SQL statement is sent in reverse polish notation. For further summary on RDA and other vendors systems see APPENDIX-A.

3.0 OPEN SQL APPROACH

The OPEN SQL goal is to provide a standard interface to NonStop SQL, DB2, SYBASE, IBM OS2 (exetended), ORACLE, INFORMIX, and many other vendors, so that the applications can access data from any of the above databases.

RDA standards are still evolving. RDA is defining protocol standards for remote server invocation, transaction management services, 2-phase commit protocols and committed to inherit ANSI/ISO SQL. But, the SQL part is not the primary goal for RDA. It comes under specialized services section to RDA. These standards are new to the industry. The process of, RDA, defining a standard is progressing slowly.

On the otherhand, IBM defined LU 6.2 provides remote server invocation, transaction management, message protocols to send and receive data, and maintenance functions services. This product is in the industry for some time, and well defined. The Distributed Transaction Processing - Transaction Processing Protocol Specification (DTP/TPPS) by OSI committee is defining a standard which is similar to IBM's LU 6.2. Some day DTP/TPPS and RDA will converge to one standard for the above services. Even if we adopt to LU 6.2 today, in the future, it would not be too difficult to adopt to the standards being defined by RDA and DTP/TPPS as there is an almost one to one mapping.

Unlike other vendors, OPEN SQL will take standards defined by the ANSI SQL committee and RDA. Wherever the definitions are incomplete, OPEN SQL will take them from IBM SAA or the most popular scheme/model from the industry. With this approach, OPEN SQL will define a standard Application Programming Interface (API) and Formats And Protocols (FAP) which will be the lowest common denominator of the existing products. With this approach every product may need to do few enhancements to their products to support OPEN SQL.

The initial plan for OPEN SQL is to take standard DYNAMIC SQL based on ANSI SQL2 specification, standard catalogs from ANSI SQL2 specification and standard data streams from RDA, IBM SAA, and others. OPEN SQL also defines standard control blocks for Errors (SQLCA), sql descriptor area (SQLDA) to access the data at run time, and additional catalogs required by OPEN SQL. OPEN SQL will attempt to use the transaction mechanism of IBM's LU 6.2 to support transaction commit, abort, resolution and remote server invocation.

4.0 HOW TO STANDARDIZE OPEN SQL?

To standardize OPEN SQL first we validate our definition by developing a prototype. Then we will promote this definition (API & FAP) among other vendors by demonstrating the working model. To promote this definition among other vendors we need to have a forum of vendors who are intereseted in participating in this kind of open standard. In that forum, we will present the OPEN SQL definition and share the views of other vendors, form a vendor committee and commit for the completion of this definition. Once an aggreement is reached by all the parties this standard needs to be published and all the participating vendors need to implement this standard. Once a standard is defined, the users and the third-party software houses can benifit from such a standard for writting portable applications and applications that can access heterogeneous databases.

4.1 PROTOTYPE

As part of the effort to standardize OPEN SQL, Tandem is developing a prototype using the proposed standards. This prototype consists of a client process and a server process. The client process is like an application sending SQL queries to access data and receives the data from the server. The server process is like a database server.

The client process accepts SQL queries from the terminal. The queries can be routed to local database or to a remote database. The local database access is provide using DYNAMIC SQL that can be understood by the local server. The remote database access is provide using OPEN SQL protocols and formats.

The server process will be implemented on a Tandem machine using NonStop SQL. The server process understands the OPEN SQL synatx, formats and protocols. The OPEN SQL synatx will be converted to NonStop SQL syntax by the server wherever necessary. The server process also converts the NonStop SQL error messages to the OPEN SQL standard before sending them to the remote client. Both the client and server exchange the data and information in the OPEN SQL format.

The client will be coded for the LXN (Tandem UNIX machine) and PC which accesses Informix, Oracle, and OPEN SQL. The client will be coded in C. The server will be coded in C on the Tandem machine. The LXN implementation uses X.25 as the communications mechanism and PC implementation use the NetBios as the communications mechanism. A "simulated" LU 6.2 interface will be programmed that works on NetBios, X.25, and Tandem Message System.

In addition to SQL commands the client process will support a MOVE command which can be used to Import and Export tables from the native data manager and the OPEN SQL server.

5.0 PROBLEMS IN MAKING OPEN SQL A STANDARD ?

OPEN SQL will only be a success if several major vendors adopt and support an OPEN SQL server which accesses their product. This is a major problem in standardizing OPEN SQL. ISO RDA is also attempting to define a standard similar to this nature. We need to influence the RDA group so that our effort and their effort would not go waste. Hence, collaboration with other vendors will be a major part of this effort. We need to attack the problem not only with technical merits but also with marketing benifits.

6.0 OPEN SQL

The OPEN SQL is a standard for accessing data from heterogeneous SQL systems. The standard consists of

a. An Application Programming Interface (API) to give applications portability.

b. A Formats And Protocols (FAP) to specify the message interface between heterogeneous SQL systems.

6.1 APPLICATION PROGRAMMING INTERFACE (API)

API defines syntactic extensions to ANSI/ISO SQL to allow an application to address multiple SQL systems. It also defines Network naming, Catalog definitions, Error reply structure (SQLCA), SQL Descriptor Area (SQLDA) to receive the number of variables, their types, and lengths in a prepared statement and other SQL syntaxes which are not defined or vague in ANSI/ISO SQL2 document.

6.2 FORMATS AND PROTOCOLS

FAP defines Message formats for SQL requests and replies will be specified along with the messages for connection to an SQL server. The data types, lengths, and their representations are also defined in FAP. The formats for data streams that flow between a client and a server are also defined. These protocols will assume the server invocation mechanism and transaction mechanism of LU 6.2.

7.0 OPEN SQL FUNCTIONAL SPECIFICATION

7.1 API

7.1.1 DYNAMIC SQL

OPEN SQL uses DYNAMIC SQL as defined in ANSI SQL X3H2 document. The ANSI SQL X3H2 lacks the definition for DESCRIBE INPUT dynamic sql which is used to get the names, data types, and data lengths of the input variables that are used in a prepared SQL statement. Tandem is proposing this enhancement to ANSI committee. See APPENDIX-C for the proposal.

ANSI SQL X3H2 definition for SQL error messages is insufficient. Tandem is proposing an enhancement to the ANSI SQL committee to include SQLCA structure for error message formats. See APPENDIX-C for the Tandem proposal on this.

7.1.2 OPEN SQL SERVER NAME

A OPEN SQL server is designated to satisfy the database service requests originated at Clients. From the Client standpoint, the server may either be remote one or a local one (at the same node as Client). Fo the user, however, the distinction does not exist. User addresses these server by a Server Name. A Server Name is a logical name which identifies a server in the network uniquely. The Datatype for Server Name is a character string. The Client invokes a server using OPEN SQL verb ALLOCATE <Server Name>, delinks the server using the verb DEALLOCATE <Server Name>. The Client can allocate more than one serever in a execution session. Client can switch servers using USE SERVER <Server Name> or Client can address a server in a DYNAMIC SQL statement using the verb AT SERVER <Server Name>.

7.1.3 SQL VERBS

```
    ALLOCATE <sql server name>

    USE SERVER <sql server name>

    PREPARE <statement name> FROM :host_variable
                               [ AT SERVER <sql server name> ]

    DESCRIBE [INPUT] <statement name> INTO :<sqlda>
                               /* default is output */

    EXECUTE <statement name> [USING [DESCRIPTOR] :host_variable]
```

```
EXECUTE IMMEDIATE <sql statement>
                        USING [DESCRIPTOR] :host_variable
                        [ AT SERVER <sql server name> ]

DECLARE CURSOR <cursorName> FOR <sql statement name>


OPEN CURSOR <cursorName> [USING [DESCRIPTOR] :host_variable]

FETCH <cursorName> {INTO ... | USING ... }

UPDATE .... WHERE CURRENT OF <cursorName>

DELETE .... WHERE CURRENT OF <cursorName>

CLOSE <cursorName>

DEALLOCATE <sql server name >
```

ANSI standard specifies that a host-variable can be used in place of <cursorName> and <statementName>. OPEN SQL adopts this standard.

7.2 OBJECT NAMES

This section discusses issues related to OPEN SQL object names, and describes a naming convention for OPEN SQL Database names and object names.

ANSI/ISO table and other related object names have the format :

```
<authorizationID>.<table ID>
```

where <authorizationID> and <tableID> are of SQL type CHAR of length 18. The <authorizationID> describes the owner of the table.

ANSI and RDA do not address the issue of naming objects in a network environment. For OPEN SQL this is a requirement as the OPEN SQL user will need to refer to the objects at a remote site. In a Tandem network 'physical names' are used for referring to remote or local SQL tables. The Tandem format is --

```
<node>.<volume>.<subvolume>.<tableName>
```

where each component is a 8 characters, and <subvolume> is a user choosen directory name.

NonStop SQL does not have logical table names like ANSI/ISO. Other vendors support only logical names and almost all of them confirm to ANSI names. Some of the vendors who support distributed access defined the object name format in the network environment and most of them support a similar format. Some of the formats are --

```
ORACLE   : <tableName>:<protocol>:<phone#>:<dbName>:...
Informix: <dbName>.<owner>.<tableName>
Sybase   : <dbName>.<owner>.<tableName>
RTI      :  <node>::<dbName>::<owner>::<tableName>
Gupta    :  ??
R*       :  <owner>.<tableName>@<dbName>
RDA      :  -- NONE --
```

OPEN SQL adopts the format <dbName>.<owner>.<tableName> for a network object. Following the ANSI/ISO standard the data types are

```
OWNER        - CHAR(18)
TABLENAME    - CHAR(18)
```

The valid characters are alphanumeric characters and a special character '_' with first character being an alpha character.

Database name is a logical name representing a unique id for the catalogs that the user is accessing in a network. ISO/RDA does not address the issue of database name(s), surprisingly. Some vendors supporting network databases or single database system have a database name concept and they have defined the SQL type for a database name. Some vendors support multiple databases per node. So <dbName> incorporates <nodeName> and <dbName>. Lets make this network database name as <NdbName>. Vendors who support database names defined the database name SQL char type. OPEN SQL defines the database name to be of the type VARCHAR (255).

```
<NdbName>     -   VARCHAR (255).
```

Any alphanymeric characters are allowed with first character being an alpha character.

OPEN SQL proposes SYNONYMS as in IBM's SAA to map fully qualified ANSI names to brief/simple names. SYNONYMS are maintained at each site in a user catalog. A user or an application can decide what synonyms it is going to use and create them using a command like:

```
CREATE SYNONYM <name> <ansiName>
```

See the CATALOGS section for the description of the SYNONYM table.

7.3 DATA TYPES

Every vendor supports common data types like CHARACTER, and INTEGER, but their lengths, precison, scale, and representations are different. Some vendors support special data types. In some machines numbers are represented left to right in 2's compliment. In some machines numbers are represented right to left. In some systems the code for integer datatype is 452, in some other systems it is 42, and some other systems it is something else. For heterogeneous data bases, there needs to be a standard set of data types, a standard representation of data and standard codes for these data types. The standard codes will make the applications understand, at run time, the data type that the application is receiving across the wire.

ANSI and ISO committees have defined standard data types. OPEN SQL inherits as much as possible these data types and the associated codes. Wherever a definition is missing or incomplete OPEN SQL will adopt the IBM SAA or a popular design. ANSI/ISO did not define the data representation, so OPEN SQL will, taking the most common representation among the vendors.

Following are the OPEN SQL data types --

1) CHARACTER (<len>) -
   specifies characters of fixed length <len>.
   Characters are entered as <u><character representation></u>.
   The Maximum value for length be 2GB.

2) VARCHARACTER (<len>) -
   specifies characters of varying length of maximum length <len>. Characters are entered as <u>&lt;character representation&gt;</u>
   The Maximum value for length be 2GB.

3) NUMERIC (<precision>, <scale>) -
   specifies the data type exact numeric, with precision and scale specified by the <precison> and <scale>. This exact numeric data type is represented in 2's complement and in left to right fashion. Numeric value is entered as
   [+/-]<unsigned integer digits>.<unsigned integer digits>.
   The maximum precision is 18 and scale is 18.

4) DECIMAL (<precision>, <scale>) -
   specifies the data type exact numeric, with scale specified by the <scale> and with implementor-defined precision greater than or equal to the value of the specified <precision>. This exact numeric data type is represented in a packed decimal form where the digits are represented left to right. This representation is the IBM SAA representation. Decimal numbers are entered as
   [+/-]<unsigned integer digits>.<unsigned integer digits>

5) INTEGER -

specifies the data type exact numeric, with scale 0. This exact numeric data type is represented in 2's complement and in left to right fashion. Integer numbers are entered as [+/-]<unsigned integer digits>. The maximum value for <unsigned integer digists> can be 2**32 - 1.

6) SMALLINT -
specifies the data type exact numeric, with scale 0 and precision smaller than the precison of INTEGER. This exact numeric data type is represented in 2's complement and in left to right fashion. Small Integer numbers are entered as [+/-]<unsigned integer digits>. The maximum value for <unsigned integer digists> can be 2**16 - 1.

7) DATETIME -
specifies the data type datetime with 7 datetime fields consisting of YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, and FRACTION of second. The limits on the values of these fields are -

```
0 <= YEAR          <= 9999
1 <= MONTH         <= 12
1 <= DAY           <= 31 (actual value is based on
                            the month)
0 <= HOUR          <= 23
0 <= MINUTE        <= 59
0 <= SECOND        <= 59
0 <= FRACTION(n)   <= 0.9(n)  where n is the number of
                              digits specified by the
                              precision of <fractional
                              seconds precision>.
                              (see ANSI X3H2 )
```

The representation of this field is GMT, Julian date based timestamp in microseconds. The date is entered as [[<year value> -]<month value> -] <days value>.
Time is entered as
[[<hours value>:]<minutes value>:]<seconds value>[.<seconds fraction>

8) REAL -
specifies the data type approximate numeric, represented using IEEE format floating point numbers.
The real number is entered as <mantissa>E<exponent> where <mantissa> is <exact numeric> and <exponent> is <signed integer>.

The codes associated with each data type are --

CHARACTER	-	1
NUMERIC	-	2
DECIMAL	-	3
INTEGER	-	4
SMALLINT	-	5
REAL	-	7
DATETIME	-	9
VARCHARACTER	-	12

7.4 NULLS

NULL - is a representation for a missing value in a column. It is not 0 for numeric data types and not 'blank' for character data types. NULL value is indicated in the user data as an indicator variable in the SQLDA structure. See SQL X3H2 specification for the description of SQLDA structure.

7.5 CATALOGS

In OPEN SQL there are two basic requirements for access to catalogs.

1. A user may need to find information about the tables, columns and other objects that can be accessed.

2. An application may need to find the characteristics of a table that the user is trying to access.

to perform joins, to use in the presentation layers of an application, and other application requirements.

ANSI/ISO has defined 6 catalog tables as a standard. They are

1) ACCESSIBLE_TABLES
2) TABLE_PRIVILEGES
3) ACCESSIBLE_COLUMNS
4) COLUMN_PRIVILEGES
5) CONSTRAINT_DEFS
6) CONSTRAINT_COLUMNS

The OPEN SQL adopts these catalog tables and adds catalog tables for SYNONYMS, LINKS, PASSWORDS, and ACCESSIBLE_DATABASES.

Following SAA, a SYNONYMS table is a catalog table maintained at each server which maps user specified table name to a table name known to the server.

The PASSWORDS table is catalog table maintained at the server end, which contain list of encrypted passwords using some one-way function. These passwords are maintained on user basis, retrieved and checked at the allocate or attach time by the OPEN SQL server.

The LINKS table is a catalog table maintained at the client side which maps database names to physical line and other network related characteristics. An example is, given a database name find the associated communication line number, protocol, speed, associated server(??), or other Network manager attributes.

The ACCESSIBLE_DATABASES table is a catalog table either maintained or syntesized at the server end. Querying this table gives all the database names accessible to user in that node/network. The implementation of this table is left to the implementor.

These catalog tables are available for each database.

The structure of each catalog is --

```
ACCESSIBLE_TABLES        (ansi)
         owner           character (18) NOT NULL
         table_name      character (18) NOT NULL
         table_type      character ( 1) NOT NULL /* T - table,V - view */

TABLE_PRIVILEGES         (ansi)
         grantee         character (18) NOT NULL
         owner           character (18) NOT NULL
         table_name      character (18) NOT NULL
         select_priv     character ( 1) NOT NULL  /* G , Y , N     */
         insert_priv     character ( 1) NOT NULL  /* G , A , S , N */
         delete_priv     character ( 1) NOT NULL  /* G , Y , N     */
         update_priv     character ( 1) NOT NULL  /* G , A , S , N */
         reference_priv  character ( 1) NOT NULL  /* G , A , S , N */

ACCESSIBLE_COLUMNS       (ansi)
         owner                character (18) NOT NULL
         table_name           character (18) NOT NULL
         column_name          character (18) NOT NULL
         data_type            character (18) NOT NULL
         data_length          integer  NOT NULL DEFAULT (0)
         data_precision       smallint NOT NULL DEFAULT (0)
         data_scale           smallint NOT NULL DEFAULT (0)
         datetime_precision   character (18) NOT NULL DEFAULT (' ')
         nullable             character ( 1) NOT NULL  /* Y , N */

COLUMN_PRIVILEGES        (ansi)
         grantee         character (18) NOT NULL
         owner           character (18) NOT NULL
         table_name      character (18) NOT NULL
         column_name     character (18) NOT NULL
         grantor         character (18) NOT NULL
         insert_priv     character ( 1) NOT NULL  /* G , Y , N */
         update_priv     character ( 1) NOT NULL  /* G , Y , N */
         reference_priv  character ( 1) NOT NULL  /* G , Y , N */

LINKS                    (open sql)
         database_name     varcharacter (256) NOT NULL
         link_attributes   varcharacter (256) NOT NULL
                                /* implementor defined */
```

```
SYNONYMS                    (open sql)
        userid              character    (18)  NOT NULL /*synonym creatorID*/
        synonym             character    (18)  NOT NULL
        owner               character    (18)  NOT NULL
        table_name          character    (18)  NOT NULL
        database_name       varcharacter (256) NOT NULL

PASSWORDS                   (open sql)
        userid              character (18) NOT NULL
        password            character (18) NOT NULL  /* encrypted */

ACCESSIBLE_DATABASES        (open sql)
        databasename        varcharacter (256) NOT NULL
```

There may be situations where a client machine does not have a server. In such cases, the LINKS table is not a SQL table but it is implementor defined.

In SYNONYMS table, the synonym will be a name (minimally qualified table name).

7.6 TRANSACTIONS (LU 6.2)

A Transaction is a atomic and durable unit of work. Transactions are initiated by a client process to safegaurd the integrity of the data. A transaction can involve a single database system or a network of database systems. A transaction whose processing requires the services of more than one transaction manager (either local or remote) is called a distributed transaction. To commit or abort transactions involving more than one database requires two-phase commit protocols. LU 6.2 provides such facility in cooperation with native transaction managers. In addition, LU 6.2 supports interprogram communication between two or more programs, such that:

1. A program can invoke a program that is on a remote site using ALLOCATE verb.

2. Programs can be distributed among multiple nodes within a network.

3. Authorization to access the data can be checked by the remote server.

4. Sessions and conversations can be started between two programs.

5. Two programs can Send and Receive data.

6. Transactions can be started automatically and commit or abort

the transactions successfully.

7. In doubt transactions can be resolved.

8. Programs can communicate using standard messages.

For further details on LU 6.2 please see APPENDIX-B.

There needs to be a API standard for transaction management. ANSI/ISO SQL has defined a standard. OPEN SQL inherits this standard. The BEGIN WORK cluase is not defined by ANSI/ISO. The three SQL verbs for transaction management are:

1. BEGIN WORK

2. COMMIT WORK

3. ROLLBACK WORK

The implementation of these verbs in the network environment will follow the LU 6.2 standard.

7.7 AUTHORIZATION

7.8 DATA STREAMS

A Client or an Application Program submits a SQL command or a Server command. It then can receive one or more of the following:

1. Command Completion Status

2. Error Messages

3. Processing Status Information

4. Rows of data

A program (client) will submit a command (SQL or non-SQL). It then receives results for the command. In general, a command will return completion status, error status, rows of data, or rows of data and completion status. In this section, we will define data streams which contain one or more of the above items. The application program needs to understand these data streams to get the correct information.

The data stream consists of header describing the information that follows the header. A header can be shared by more than one row of data. Such a header is called shared header. On the other hand, there can be header for each row describing the contents of the row. Such a header is called header. A header can also describe additional information like erro information and command completion status.

These formats can be best described as :

SHARED HEADER DESCRIPTION:

```
+----------------+---------+--------------+---+--------------+
|shared|length of|number of|data|offset in |...|data|offset in |
|header|header   |rows     |type|the buffer|   |type|the buffer|
+----------------+---------+--------------+---+--------------+
   <integer>      <integer>   <integer>         <integer>
```

DATA BUFFER LAYOUT:

```
+--------------+----+----+---------------+--------------+
|data|NULL     |data|data|...............|data|NULL     |
|COL1|indicator|COL2|COL3|               |COLn|indicator|
+--------------+----+----+---------------+--------------+
  <integer>     <INT><INT>                 <integer>
```

HEADER DESCRIPTION:

```
+----------------+---------+--------------+---+--------------+
|header|length of|number of|data|offset in |...|data|offset in |
|      |header   |rows=1   |type|the buffer|   |type|the buffer|
+----------------+---------+--------------+---+--------------+
   <integer>      <integer>   <integer>           <integer>
```

DATA BUFFER LAYOUT:

```
+--------------+----+----+---------------+--------------+
|data|NULL     |data|data|...............|data|NULL     |
|COL1|indicator|COL2|COL3|               |COLn|indicator|
+--------------+----+----+---------------+--------------+
   <integer>    <INT><INT>  <integer>        <integer>
```

RESPONSE BUFFER DESCRIPTION:

```
+----------------+----------------------------------------------+
|SQLCA |length of|Error message in the SQLCA format.            |
|buffer|Message  |All the buffer pointers are converted to offsets.|
+----------------+----------------------------------------------+
   <integer>
```

Shared Header and Header describe the contents of the row(s)of data that are following. Header is used when each row content is different. Shared Header is used to describe the contents of a row when more than one row of data share the same description. For example, when 100 rows have the same format a shred header is used. For example, if row 1 has different layout from row 2 then there will be one header for row 1 and a separate header for row 2. In most of the cases a shred header is sufficient. The total length of the row is determined by subtracting the total length of the header from offset in the last column and adding 1. For example, if the header and row are of the form:

```
+=================================+--------------------------+
|SH|20 | 10 | x|24 | y|34 | z|37 |COL1|    COL2          | COL3|
+=================================+--------------------------+
```

The interpretation for this buffer is:

o This is shared header.

o There 10 rows sharing this header.

o There are three columns in each row {(((20-4)/4)-1) = 3}.

o The total length of the row is (37-20+1) = 18 bytes.

o Type of COL1 is x, COL2 is y, CO13 is z.

o Length of COL1 is 24-20+1 = 5.

o Length of COL2 is 34-24 = 10 (Notice the difference between COL1 & 2).

o Length of COL3 is 37-34 = 3.

The interpretation for Header will be same except that the number of rows field will be 1 always. The length of each column includes the space for the NULL indicator variable. The length of this field is 1. If the column is NULL the indicator variable field will contain a positive value else it will be zero. The client should map this data buffer to SQLDA structure in the user application area specially the SQLDATA POINTER and SQLIND POINTER.

The Response Buffer Description is simple. It contains a token indicating that this is a SQLCA message buffer. The second field in the header will indicate the total length of the SQLCA structure that is being returned and the rest of the buffer contains the SQLCA structure as described in the APPENDIX-C.

The SQLCA structure can contain positive responses, acknowledgements, error messages, and command completion codes.

7.9 CALL INTERFACE TO OPEN SQL

8.0 FUTURE WORK

The defintion of OPEN SQL presented in this paper address the issue of an application accessing data from multiple heterogeneous database systems. This defintion uses DYNAMIC SQL only. The possible future work in the OPEN SQL is -

- o Support of precompiled SQL, to improve the run time performance of the applications.

- o Allow object names belonging to more than one heterogeneous database in a single SQL query. This involves the syntax definition, semantic definition and the consequences in the servers.

- o A standard for query execution costs and/or statistics associoted with objects.

9.0 FUNDS TRANSFER EXAMPLE USING OPEN SQL

```
/* ------------------------------------------------------- */
/* Following is an example using OPEN SQL. This program    */
/* is written in C language. The code fragments shown below*/
/* demonstrates a fund transfer between a new york branch  */
/* and san francisco branch. The code gets the account     */
/* numbers for new york and san francisco branches         */
/* and the amount to be transfered from new york account   */
/* to san francisco account. Gets the account number and   */
/* balance from nyc; if account exists and balance is not  */
/* going to be negative then subtracts the amount from nyc,*/
/* adds the same amount to sfo and logs this information   */
/* in a history file. The transaction is commited at the   */
/* successful conclusion. The transaction is abort if some */
/* event is not successful in this process.                */
/* ------------------------------------------------------- */

/* ------------------------------------------------------- */
/* Include SQL structures to be used in the sample program */
/* ------------------------------------------------------- */
EXEC SQL INCLUDE SQLDA END-EXEC;
EXEC SQL INCLUDE SQLCA END-EXEC;

/* ------------------------------------------------------- */
/* Define variables to be used in the sample program.      */
/* ------------------------------------------------------- */
char      *NYCserver;
char      *SFOserver;

char      *selectStmt;
char      *creditStmt;
char      *debitStmt;
char      *histStmt;

char      *nyc_account;
char      *sfo_account;

char      *nycacct;
char      *sfoacct;

float      nyc_balance;
float      xfer_amount;

struct SQLDA    *SQLDASelInput;
struct SQLDA    *SQLDASelOutput;
struct SQLDA    *SQLDAdebInput;
struct SQLDA    *SQLDAcreInput;
struct SQLDA    *SQLDAhistInput;

/* ------------------------------------------------------- */
/* Inititalize varaibles to hold SQL strings to be         */
/* PREPAREd at NYC and SFO locations.                      */
/* ------------------------------------------------------- */


/* ------------------------------------------------------- */
/* Initialize the server names etc and read user parameters*/
/* ------------------------------------------------------- */
strcopy (NYCserver, "NYCOPENSQL");
strcopy (SFOserver, "SFOOPENSQL");

read_input_from_user (nycacct, sfoacct, xfer_amount);


/* ------------------------------------------------------- */
/* Start a OPEN SQL server at the NYC node.                */
/* ------------------------------------------------------- */
EXEC SQL ALLOCATE NYCserver END-EXEC;
if (sqlca.sqlcode != 0)
   { /* display unable to allocate server */
   goto error;
   }

/* ------------------------------------------------------- */
/* Initiate a transaction                                  */
/* ------------------------------------------------------- */
EXEC SQL BEGIN WORK END-EXEC;

/* ------------------------------------------------------- */
/* Prepare the select statement at NYC                     */
/* ------------------------------------------------------- */
strcopy (selectStmt,
"select NYCaccount,NYCbalance INTO nyc_account,nyc_balance "&
"       FROM NYCBRANCH WHERE NYCaccount = :nycacct");

EXEC SQL PREPARE sel   FROM :selectStmt END-EXEC;
if (sqlca.sqlcode != 0)
   { /* display error */
   goto error;
   }

/* ------------------------------------------------------- */
/* Prepare a update statement at NYC                       */
/* ------------------------------------------------------- */
strcopy (debitStmt,
"update NYCBRANCH set NYCbalance = NYCbalance - :xfer_amount"&
"       WHERE NYCaccount = :nycacct");

EXEC SQL PREPARE debit FROM :debitStmt END-EXEC;
if (sqlca.sqlcode != 0)
   { /* display error */
   goto error;
   }

/* ------------------------------------------------------- */
/* Prepare a update statement at SFO with an implicit      */
/* SFOserver ALLOCATION.                                   */
/* ------------------------------------------------------- */
strcopy (creditStmt,
"update SFOBRANCH set SFObalance = SFObalance + :xfer_amount"&
"       WHERE SFOaccount = :sfoacct");

EXEC SQL PREPARE credit FROM :creditStmt AT SFOserver END-EXEC;
if (sqlca.sqlcode != 0)
   { /* display error */
   goto error;
   }

/* ------------------------------------------------------- */
/* Prepare a insert statement at NYC                       */
/* ------------------------------------------------------- */
strcopy (histStmt,
"insert into HISTORY values "&
"(:nycacct, :sfoacct, :xfer_amount, CURRENT_TIMESTAMP)");

EXEC SQL PREPARE hist FROM :histStmt END-EXEC;
if (sqlca.sqlcode != 0)
   { /* display error */
   goto error;
   }

/* ------------------------------------------------------- */
/* Get the input parameters and its attributes w.r.t select*/
/* ------------------------------------------------------- */
EXEC SQL DESCRIBE INPUT sel   INTO :SQLDASelInput END-EXEC;
/* ------------------------------------------------------- */
/* Get the output params and its attributes w.r.t select   */
/* ------------------------------------------------------- */
EXEC SQL DESCRIBE       sel   INTO :SQLDASelOutput END-EXEC;

/* ------------------------------------------------------- */
/* Get the input parameters and its attributes w.r.t update*/
/* ------------------------------------------------------- */
EXEC SQL DESCRIBE INPUT debit INTO :SQLDAdebInput END-EXEC;

/* ------------------------------------------------------- */
/* Get the input parameters and its attributes w.r.t update*/
/* ------------------------------------------------------- */
EXEC SQL DESCRIBE INPUT credit INTO :SQLDAcreInput END-EXEC;

/* ------------------------------------------------------- */
/* Get the input parameters and its attributes w.r.t insert*/
/* ------------------------------------------------------- */
EXEC SQL DESCRIBE INPUT hist   INTO :SQLDAhistInput END-EXEC;

/* ******************************************************* */
/* Fill in the SQLDA buffers with the information like     */
/* nyc account number, xfer amount, sfo account number,    */
/* make SQLDA point to local variables etc., and           */
/* EXECUTE select statment at NYC.                         */
/* ******************************************************* */
EXEC SQL EXECUTE sel   USING :SQLDASelOutput END-EXEC;
```

```
if (sqlca.sqlcode < 0)
   { /* display error */
   goto error;
   }
if (sqlca.sqlcode == RECORD_NOT_FOUND)
   {/* display account cancelled */
   goto error;
   }
else
   {
   if (nyc_balance - xfer_amount < 0)
      {/* disaply insufficient funds */
       goto error;
      }
   else
     {
     /* ------------------------------------------ */
     /* Subtract xfer amount from NYC account.     */
     /* ------------------------------------------ */
     EXEC SQL EXECUTE debit USING :SQLDAdebInput END-EXEC;
     if (sqlca.sqlcode != 0)
        {/* failed to complete the transaction */
        goto error;
        }

     /* ------------------------------------------ */
     /* Add      xfer amount to   SFO account.     */
     /* ------------------------------------------ */
     EXEC SQL EXECUTE credit USING :SQLDAcreInput
                             AT SERVER SFOserver END-EXEC;
     if (sqlca.sqlcode != 0)
        {/* failed to complete the transaction */
        goto error;
        }

     /* ------------------------------------------ */
     /* Insert this information to HISTORY file@NYC*/
     /* ------------------------------------------ */
     EXEC SQL EXECUTE hist   USING :SQLDAhistInput END-EXEC;
     if (sqlca.sqlcode != 0)
        {/* failed to complete the transaction */
        goto error;
        }

     /* ------------------------------------------ */
     /* Make the updates permanent.                */
     /* ------------------------------------------ */
     EXEC SQL COMMIT WORK END-EXEC;
     /* display transaction completed successfully */
     return (success);
     }
   }

error:
```

```
/* ------------------------------------------ */
/* Undo the changes done sofar                */
/* ------------------------------------------ */
EXEC SQL ROLLBACK WORK END-EXEC;
}
```

10.0 REFERENCES

1. Antoni Wolski - LINDA: A System for Loosely Integrated Databases
2. OSI - Documnet #8825: Specification of Basic Encoding Rules for Abstract Syntax Notation.
3. OSI - Document #ISO/TC/97/SC21: Remote Database Access, Tutorial.
4. OSI - Document #ISO/RDA/WD87: Third Working Draft of ISO Remote Database Access.
5. OSI - Document #ISO/TC/97/SC 21: Information Processing Systems - Distributed Transaction Processing.
6. ANSI - Document #ANSI X3H2-87-210 ANSI SQL standard.

11.0 APPENDIX - A

VARIOUS VENDORS AND SUMMARY OF THE SYSTEM

11.1 RDA

Remote Database Access (RDA) is a working draft OSI document.

* It depends on

ROSE (remote procedure call)

CCR (Commit, Concurrency control, Recovery)

These are in vague shape. We use LU 6.2 terminology in preference.

RDA is generic, it could apply to any database language. The generic parts are just the LU6 ATTACH and SYNC logic.

It is specialized for ANSI SQL as follows:

1. Dynamic SQL is the statement language
2. ANSI SQL catalogs (tables, columns,...)
3. OSI data stream of records comming back
4. Standard error numbers.

Summary:

1. Lets ignore OSI transactions and RPC. Rather presume LU 6.2 verbs ALLOCATE and DEALLOCATE, SEND and RECEIVE, SYNC and BACKOUT.

2. RDA-SQL has the verbs:

```
EXECUTE(conversation, sql-statement)
          returns (count, {error, [record]},...)
       This is execute immediate. The ENTIRE answer comes back
       in one buffer. AMAZING!

DEFINE (conversation, sql-statement) returns (handle);
       This is approximately the PREPARE SQL statement.

INVOKE (handle, {args},...) returns (count, {error, [record]},...)
       This is approximately EXECUTE.  Again, the entire
       answer is returned in one buffer.  Not clear how
```

OPEN, FETCH, UPDATE, DELETE via cursor work.

DROP   (handle);

The verbs are encoded in Polish-prefix notation.

The data stream comming back is encoded in <type>.<length>.<data>. Chars are fixed length ascii, Floats are IEEE float, Integers are 2s complement, Decimal is ??????????. Data types are not completely specified (e.g. null not specified).

The catalogs are defined to be the same as the ANSI SQL catalogs:

```
   TABLES(owner, table-name, creator, type, columns)
   COLUMNS(owner, table,col-name, creator, number,
           type, length, precision, scale, constraint, title);
   AUTHORIZATION-IDS(owner);
   PRIVILEGES(grantee, owner, table, select, delete, insert, update);
   UPDATABLE-COLUMNS(grantee, owner, table, column-number);
```

The errors are very generic (none related to SQL)

protocol errors
data format errors
authorization errors
transaction errors
bad-statement

Critique:

1. The dynamic SQL interface is incomplete (no DESCRIBE, SQLDA, SQLCA, SQLSA, ...). It does not follow the proposed ANSI dynamic SQL design. Suggests the designer did not understand dynamic SQL.

2. There is no obvious way to use cursors via this interface.

3. Each statement returns it's answer in one block of records. If the answer is large (say 1MB) this is likely not to work very well. There should be a continuation protocol.

4. The OSI data stream is verbose and obscure. Each item is described as <type>,<length>,<data> in a Huffman encoding of type and length.

5. The rational for sending polish rather than SQL is very questionable. It is a difficult to specify and difficult to debug interface. Not desirable for heterogeneous systems.

UPDATE TABLEX SET COLUMNB = 107 WHERE COLUMNA = "ARTICLE ABC"
is in RDF:

```
   A9563054300a050016065461626c6558301430121607436f6c756d6e423007
   8000A10301000730308000a22c30118000a40d300b05001607436f6c756d6e
   418188a11530138000a10fa20d160b41727469636c6520414243
Which would you rather debug / specify?
```

11.2 INFORMIX

Informix uses a requestor/server structure in which the requestor sends dynamic SQL and the server replies with a data stream.

The INFORMIX SQL is very close to ANSI SQL. They have the standard EXECUTE IMMEDIATE, PREPARE, DESCRIBE, EXECUTE, OPEN, FETCH, UPDATE, DELETE, CLOSE dynamic verbs. They use a SQLCA and SQLDA similar to the Dynamic SQL specified in the SQL2 working draft.

The command stream in includes a pair of OPEN <database>, and CLOSE <database> verbs verbs to connect the requestor to a specific database.

The data stream has only two types of representation:

Fixed length ASCII characters and decimal numbers with the representation <exponent>,<sign>,<length>,<value>. The first three attributes are represented as integer bytes. The value attribute is represented base 100, with one digit per byte.

11.3 SUN

SUN does not support SQL directly but they have a widely implemented data stream and remote procedure call mechanism.

TCP/IP is a session mechanism

XDR is the SUN data stream. In it all items are multiples of 4 bytes. The standard types are integer (2s complement), IEEE Float, Character (ASCII), and VarCharacter (4 byte length field).

RPC is the SUN remote procedure call mechanism.

NFS is a file system built atop RPC and XDR.

YELLOWPAGES is a name service.

From Open SQL's viewpoint, TCP/IP, XDR, and RPC is simple and widely available software. LU 6.2 could map down to TCP/IP rather than SNA as a transport mechanism. XDR might form the basis for a standard data stream. LU 6.2 can not use RPC for a variety of reasons: (rpc is datagrams, not sequenced delivery; rpc is context free which implies overhead (late binding) for each call.

11.4 ORACLE*

ORACLE uses a requestor/server structure in which requestor sends dynamic SQL and server replies with a data stream.

ORACLE(5.2a) is NOT fully ANSI compatible. Their SQL is close to IBM DB/2 SQL. However, they have standard EXECUTE IMMEDIATE, PREPARE, DESCRIBE BIND (for describe input), DESCRIBE SELECT LIST (for describe output), EXECUTE, FETCH, OPEN, UPDATE, DELETE, and CLOSE dynamic verbs. They use SQLCA, and SQLDA similar to the Dynamic SQL specified in the SQL2 working draft and exactly same as IBM DB/2. They lack the definition for SQLSA.

The command stream includes CONNECT and RELEASE verbs to connect to a specific database and release from that database.

The data stream can contain CHAR, VARCHAR, INTEGERS(1,2,4 bytes), FLOATINGPOINT(4 and 8 bytes), DECIMAL, BINARY, and DATE data types. ORACLE fixed numbers have <length><sign><exponent><digits> format (similar to INFORMIX). Length = #of digits + 1. Exponent ranges from 0 to 127. Each digit is binary number from 0 to 99. Floating point numbers are represented using hardware supported floating point numbers. DECIMAL data is in BCD form with implicit decimal point. ORACLE date datatype is a seven byte field containing year(2 bytes), month, day, hours, minutes, and seconds.

Oracle converts ORACLE data types to application variable types whenever possible.

SQL*NET provides the ability to connect to a heterogeneous node running ORACLE.

11.5 SYBASE

SYBASE uses a requestor/server structure in which requestor sends TRANSACT-SQL and the server replies with Tabular Data Stream. TRANSACT-SQL includes SQL statements, DBA commands, Transaction control statements, Procedures, and Triggers.

SYBASE SQL is not ANSI SQL. It does not have cursors. The dynamic SQL interface is incomplete (no DESCRIBE, SQLDA, SQLCA, SQLDA,...).

In addition, the programming interface is a set of procedures (DBLIB) rather than a preprocessor.

On the other hand, it provides save points, procedures which have control flow, SQL and calls to other procedures. Procedure can be compiled and stored in the catalogs for later execution.

The command stream includes DBOPEN <database>, DBCLOSE <database> procedures to connect to a database and DBLOGIN <servername> and DBEXIT (from server/session)to connect to a server.

The data stream has TOKEN representation. A TOKEN is represented by a single byte followed by token specific data. There are five classes of tokens organized by size:

(1) Zero Length
(2) Fixed Length of 1,2,4, or 8 bytes
(3) Short variable length
(4) Long Variable length
(5) Very Long Variable length.

Zero length tokens do not have data. Short variable length data has length field of one byte (maximum 255 bytes). Long variable length data has two byte length field(maximum 65538 bytes of data). Very long variable length data has four byte field length (maximum 2**31 bytes of data). NULL data is represented as short variable length with field length as zero. FLOAT data is a fixed length data of length 8 bytes. FLOAT data format is negotiated at connect. Sybase does the necessary byte-sex and numeric translation to allow heterogeneous systems to communicate. INTEGER data is a fixed length data of length 1,2, and 4 bytes. MONEY and DATETIME are fixed length data of length 8 bytes. BIT data types occupy one byte minimum. BIT data types can not have NULLs.

Summary:

SYBASE datastream is reasonable. Good for variable length data. The lack of a SQL preprocessor, extensions, lack of cursors, and minimal dynamic SQL support make it difficult to be ANSI compatible.

11.6 TERADATA

Teradata connects to IBM 370, Honeywell, and to ATT 3Bxx computers.

It attaches via Ethernet or IBM Channel.

The interface is Dynamic SQL: SQL in, records and messages out. The interface is just like DB2: Dynamic SQL with Describe but you can describe both input and output.

All messages have a header and a body. The body consists of parcels.

```
After setup,
     input parcels are SQL statements with params
     output parcels are:
         records parcels: <nulls bit vector> {<fields>}...
         SQLDA data: IVAR/OVAR descriptors
         SQLCA data (error codes and messages)
         SQLSA data (statistics)

     fields are in the host native representation:
            e.g.: Floats in IBM float, Honeywell Float, or
                  3Bxx float Chars are EBCDIC in IBM, ANSI
                  in ATT
     VARCHARS have 2-byte length fields and records have to
            fit in 32,000 bytes.
```

Teradata has the best developed of the data streams. Their design includes:

- encryption
- double buffering
- reset cursor to start of set
- abort operation

Their setup protocol is non-standard but it has the standard flavor of

```
     LOGON TO DATABASE

     SETUP DYNAMIC STATEMENT
     USE IT.
```

LOGOFF DATABASE

They have SQL preprocessors for PL/1, Cobol and soon C.

They also provide a call library (examples are in C).

There is no attempt to coordinate transaction commit between the host and the Teradata database.

11.7 TANDEM

Tandem's NonStop SQL is a fully distributed database management systems integrated into the GUARDIAN 90 operating system. Users can access and update data anywhere in a network. Tandem provides distributed transactions in a tandem network with rollback and rollforward recovery.

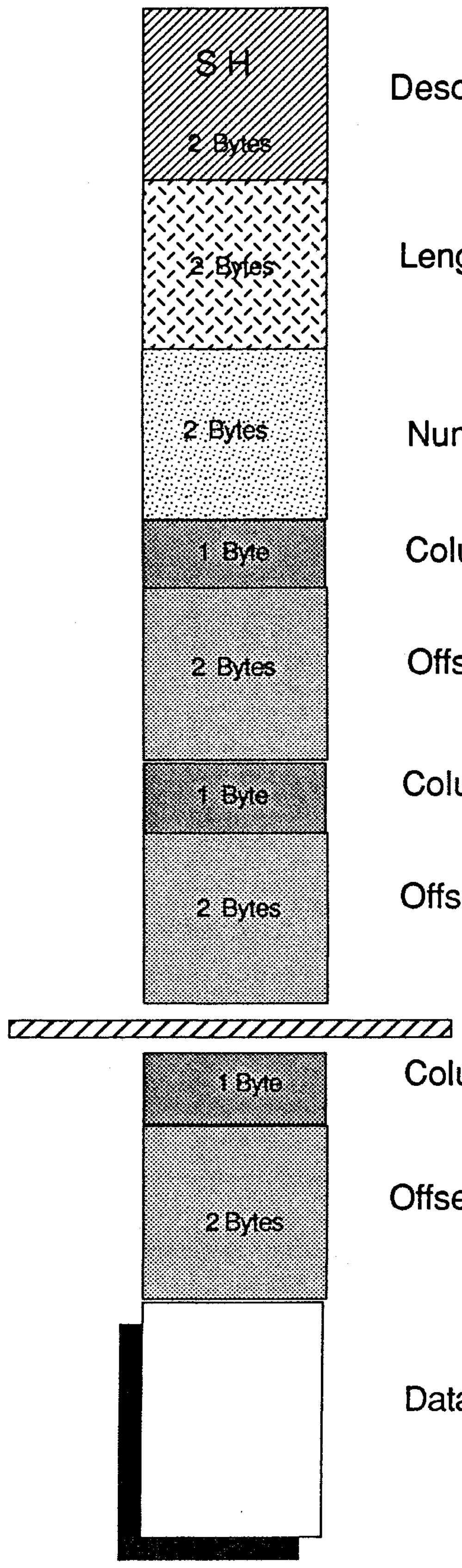
Tandem SQL is almost like ANSI SQL. It provides dynamic SQL, static SQL, and cursors. However, there are substantial extensions to SQL in the DDL,DML and DCL areas. The cursor defintion syntax is not ANSI compatible. They have standard EXECUTE IMMEDIATE, PREPARE, DESCRIBE INPUT, DESCRIBE OUTPUT, EXECUTE, FETCH, OPEN, CLOSE, UPDATE, DELETE, and INSERT dynamic verbs. They use SQLCA and SQLDA where SQLCA is pretty close to IBM SQLCA and SQLDA is close to ANSI SQLDA. In addition Tandem has SQLSA for statistics.

Tandem does not have a Database concept. Once user logs on to system he can access all the table in the network that the user is authorized to. Tandem authorization model and object names are not ANSI compatible.

The data stream can contain CHAR, VARCHAR, INTEGERS (2,4,8), DECIMAL, NUMERIC (<precision>, <scale>). Tandem does not support NULLS yet. They do not have support for DATETIME and FLOATING POINT numbers.

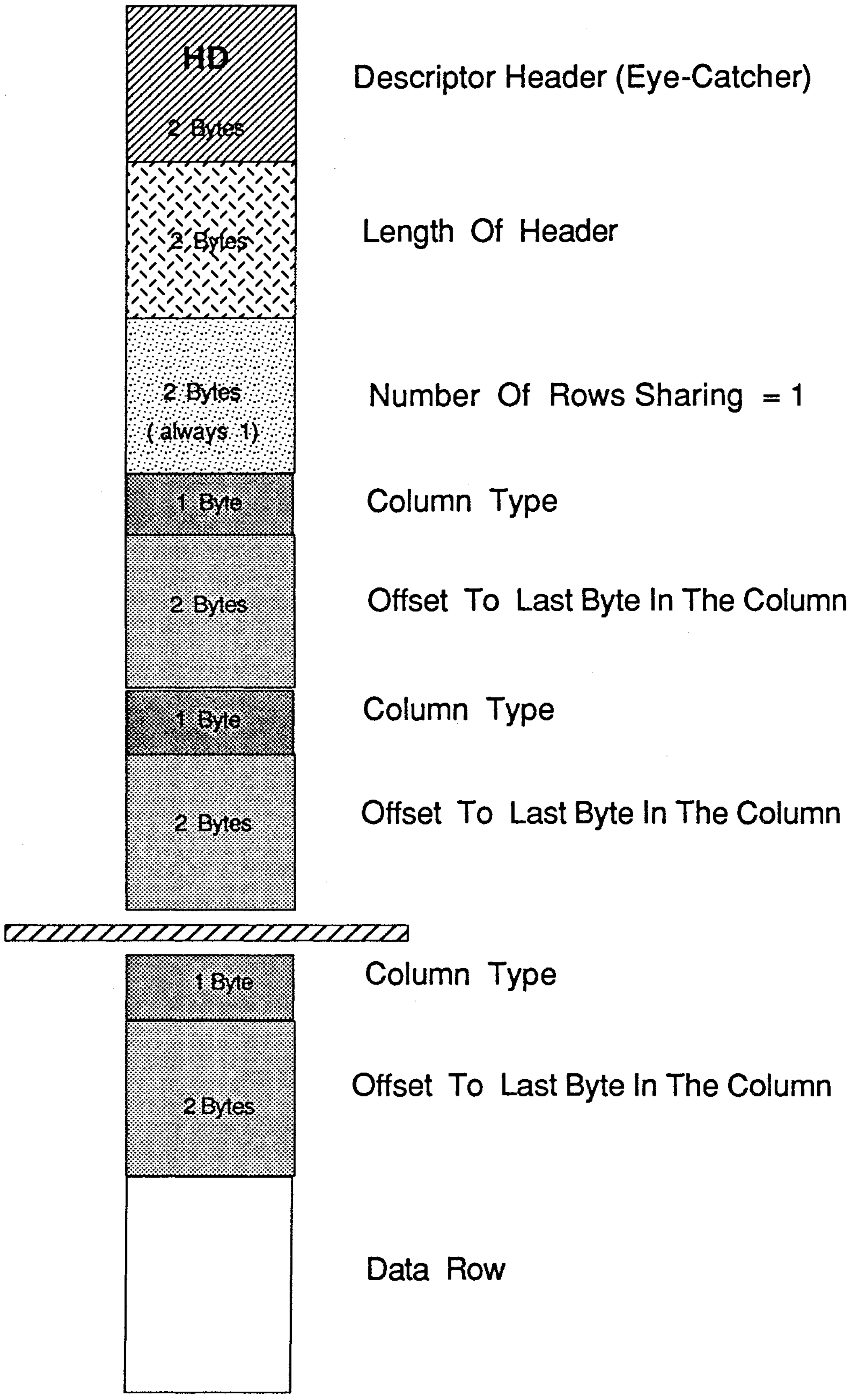
Tandem converts TANDEM data types to application variable types whenever possible.

SHARED DESCRIPTOR



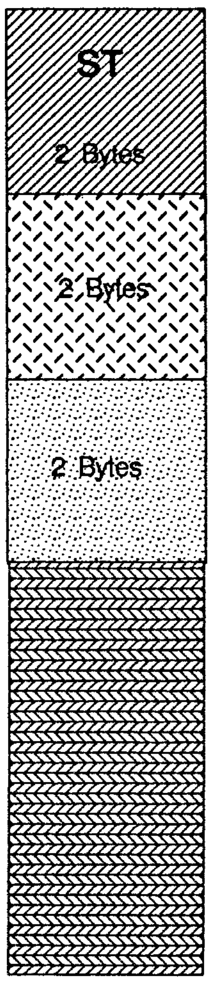
OPEN SQL - DATA STREMS

DESCRIPTOR


HD
2 Bytes
Descriptor Header (Eye-Catcher)
2 Bytes
Length Of Header
2 Bytes
(always 1)
Number Of Rows Sharing = 1
1 Byte
Column Type
2 Bytes
Offset To Last Byte In The Column
1 Byte
Column Type
2 Bytes
Offset To Last Byte In The Column
1 Byte
Column Type
2 Bytes
Offset To Last Byte In The Column
Data Row

OPEN SQL- DATA STREAMS

STATUS DESCRIPTOR



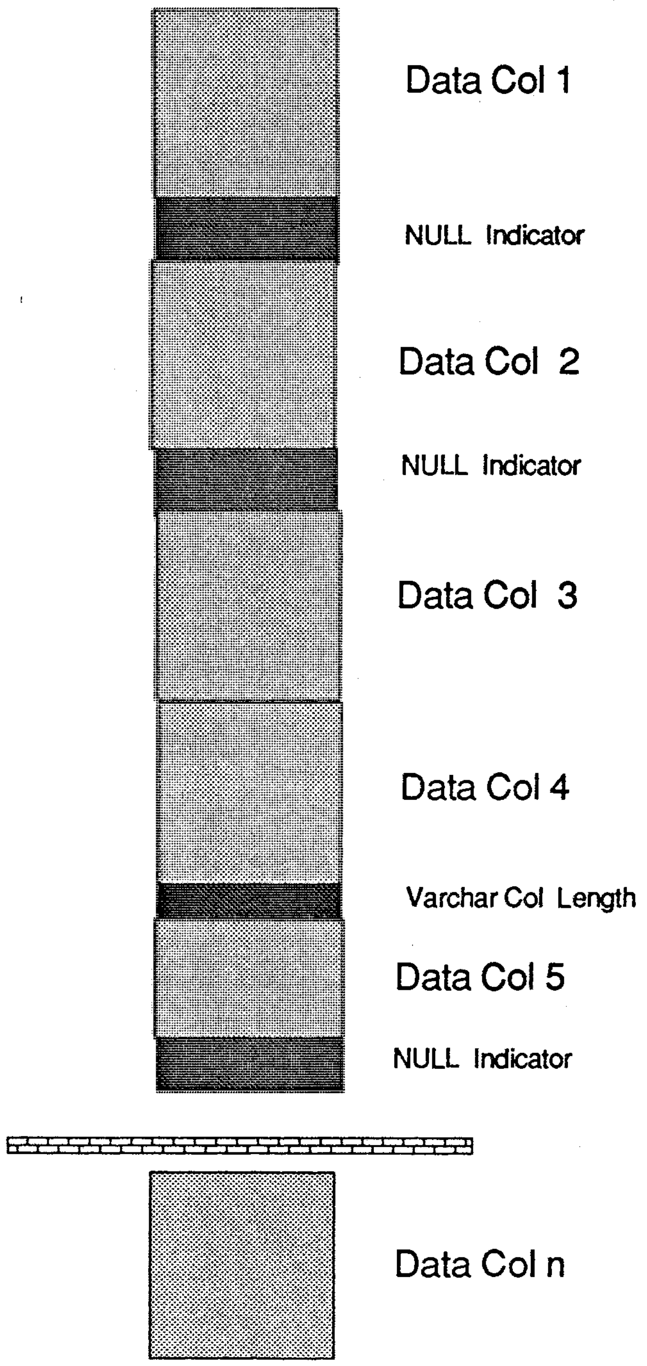
Descriptor Header (Eye-Catcher)

Status Message Type
- SQLCA
-
-

Length of Status Message

Status Message

DATA BUFFER LAYOUT



OPEN SQL - DATA STREAMS

12.0 APPENDIX - B

LU 6.2

**** TO BE FILLED IN ****

13.0 APPENDIX - C

ANSI PROPOSAL

X3H2

SQL2 CHANGE PROPOSAL.

DOCUMENT NUMBER:. X3H2-88-???

DATE:. December 20, 1987.

AUTHORS:. Amardeep Sodhi, Ramarao Yendluri.

SUBJECT:. Dynamic SQL - DESCRIBE INPUT and SQLCA.

References:. 1. X3H2-87-207: ISO-ANSI (working draft) SQL2, dated August 1987.

SUMMARY.

Dynamic SQL was proposed in reference 1. In addition to DESCRIBE statement, SQLDA was also specified. When a SQL statement is prepared it can contain host variables or params which are instantiated at execute time. In an application program using dynamic SQL the number and data types of input params or host variables are often not known until runtime. However, the SQL the run time executor knows exactly, how many input variables are specified in a prepared statement and what data types are expected for these variables or params. ISO-ANSI proposal in reference 1 lacks a way for the application program to extract this descriptive information. We propose an extenstion to the ANSI DESCRIBE statement, called DESCRIBE INPUT, by which a program can acquire the number, names and types of input host variables/params of a dynamic SQL statement.

Many vendors support a detailed structure for error codes, warnings and other related information regarding the query execution status. The ANSI proposed SQLCODE and SQLSTATE are not sufficient for most applications. Most of the applications would like to know that an error had occurred or a warning had occurred. For example, an application would like to know that the access to a view was rejected because the view definition was not found or the underlying table(s) were not found. Further more, the SQL runtime execution unit might give more than one warning/error and the user/application would like to know all such errors and/or warnings. IBM's DB2, ORACLEs SQL*, INFORMIX, TERADATA, TANDEM and many other vendors do support a similar structure called SQL Communication Area (SQLCA). In addition, for Remote Data Access (RDA) such a standard along with standardization of error and warning codes will be important.

Finally, most vendors and users are interested in knowing the cost of the execution of a particular query or group of queries. Also, the information about number of rows deleted, inserted, updated, or selected when a query is executed is important to know. We propose this information be included as a part of the SQLCA structure.

EXAMPLES:.

The following are examples (written in C language) of how the DESCRIBE INPUT dynamic statements and SQLCA can be used. The intent is not to show the complete code fragments but just to indicate the order of the dynamic statements and how they work together.

Non-SELECT Statement.

```
EXEC SQL BEGIN DECLARE;
char statement [100];
long v1,v2,v3,v4,v5;
EXEC SQL END DECLARE;

EXEC SQL INCLUDE SQLDA;
EXEC SQL INCLUDE SQLCA;

struct SQLDA   SQLDAIN;
struct SQLDA   SQLDAOUT;

strcpy (statement, "insert into EMP values (?v1,?,?,?v2)");

EXEC SQL PREPARE s1 FROM :
statement;
if (SQLCA.sqlcode == 0)
continue
else if (SQLCA.sqlcode < 0)
raise_exception
else if (SQLCA.sqlcode > 0)
look_for_more_warnings_and_raise_warnings;

/* allocate 10 sqlvar structures */
SQLDAIN.SQLN = 10;

/* Get information about input variables */
EXEC SQL DESCRIBE INPUT s1 INTO :SQLDAIN;
if (SQLCA.sqlcode != 0)
raise_error;

if (SQLDAIN.SQLD == 0) /* no input params/variables */
     EXEC SQL EXECUTE s1;
else {
        for (i=0;i < SQLDAIN.SQLD;i++)
        /*  fill in SQLDAIN information and make
            SQLDAIN.sqldata point to input variable values */
        EXEC SQL EXECUTE s1 USING :SQLDAIN;
      }

if (SQLCA.sqlcode == 0)
    continue
else if (SQLCA.sqlcode < 0)
        raise_exception
else if (SQLCA.sqlcode > 0)
        look_for_more_warnings_and_raise_warnings;
```

SELECT Statement.

```
EXEC SQL BEGIN DECLARE;
char statement[100];
char name[30];
char cusrorname[20];
EXEC SQL END DECLARE;

EXEC SQL INCLUDE SQLDA;
EXEC SQL INCLUDE SQLCA;

struct SQLDA   SQLDAIN;
struct SQLDA   SQLDAOUT;

long empid;
int i;

SQLDAIN.SQLN = 10;    /* indicate 10 sqlvar occurrences */
SQLDAIN.SQLVAR = sql; /* set pointer to sqlvar occurrences */

strcpy(cursorname, "c1"); /* set cursor name */
strcpy(statement,
"SELECT * FROM EMP WHERE EMPID = :emp_id AND LNAME = :l_name");

EXEC SQL PREPARE s1 FROM :statement;
if (SQLCA.sqlcode == 0)
continue
else if (SQLCA.sqlcode < 0)
raise_exception
```

```
else if (SQLCA.sqlcode > 0)
look_for_more_warnings_and_raise_warnings;

EXEC SQL DESCRIBE INPUT s1 INTO :SQLDAIN;
if (SQLCA.sqlcode != 0)
raise_error;
if (SQLDAIN.SQLD == 0) /* no input params/variables */
     EXEC SQL EXECUTE s1;
else {
        for (i=0;i < SQLDAIN.SQLD;i++)
        /*  fill in SQLDAIN information and make
            SQLDAIN.sqldata point to input variable values */
        EXEC SQL DECLARE c1 CURSOR FOR :s1;
        if (SQLCA.sqlcode == 0)
        continue
            else if (SQLCA.sqlcode < 0)
        raise_exception
            else if (SQLCA.sqlcode > 0)
        look_for_more_warnings_and_raise_warnings;

        SQLDAOUT.SQLN = 20;
        SQLDAOUT.SQLVARS = sq2;

        EXEC SQL DESCRIBE s1 INTO :SQLDAOUT;
        if (SQLCA.sqlcode == 0)
        continue
            else if (SQLCA.sqlcode < 0)
        raise_exception
            else if (SQLCA.sqlcode > 0)
        look_for_more_warnings_and_raise_warnings;

        if (SQLDAOUT.SQLD == 0)      /* No output variables */
           raise_error;
        else  {
              /* Allocate fetch buffer for each column */
              for (i=0;i < SQLDAOUT.SQLD;i++)
              /*  fill in SQLDAOUT information and make
              SQLDAOUT.sqldata point to output variables */

              EXEC SQL OPEN c1 USING :SQLDAIN;

              if (SQLCA.SQLCODE != 0)
                 raise_open_error;
              }
              while (SQLCA.SQLCODE == 0) {
              EXEC SQL FETCH c1 USING DESCRIPTOR :SQLDAOUT;
              if (SQLCA.SQLCODE == 0)
              {
              /* display the row contents */
              for (i=0;i<SQLDAOUT.SQLD;i++)
```

```
    printf("%s : ",SQLDAOUT.SQLVARS[i].SQLDATA);
printf ("\n");
}
else break;
}
if (SQLCA.sqlcode == 0)
   continue
else if (SQLCA.sqlcode < 0)
        raise_exception
else if (SQLCA.sqlcode > 0)
        look_for_more_warnings_and_raise_warnings;

EXEC SQL CLOSE c1;
```

PROPOSAL.

- Update the syntax for DESCRIBE statement in section 11.6 page 267 as :

```
-----------------------------------------------------------------
|                                                               |
| DESCRIBE [INPUT] <SQL statement identifier> INTO <sqlda       |
| descriptor name>                                              |
|                                                               |
-----------------------------------------------------------------
```

- Add the following sentence to SYNTAX RULES in section 11.6 page 267

3) Case

a) If INPUT is specified then the execution of the <describe statement> sets the <sqlda descriptor name> with the number of input variables, their names, and types for the prepared statement identified by <SQL statement identifier>.

b) Otherwise, the execution of the <describe statement> sets the <sqlda descriptor name> with the number of output variables, their names, and types for the prepared statement identified by <SQL statement identifier>.

- Update General Rules (3) in section 11.6 page 267 as :

3) Case

a) If INPUT is specified in the <describe statement> then the execution of the <describe statement> shall set the values of the descriptor area with information as described in Clause 11.1 (case 7), "<dynamic using clause>".

b) If INPUT is NOT specified in the <describe statement> then the the execution of the <describe statement> shall set the values of the descriptor area with information as described in Clause 11.1 (case 3 & 4), "<dynamic using clause>".

- Add case 7 to <dynamic using clause> in Clause 11.1 :

7) When <sqlda descriptor name> is used in a <describe statement> with INPUT option, the result is a description of the input variables of the prepared <dynamic SQL statement>.

The information returned in <sqlda descriptor name> shall be as follows:

a) SQLN shall contain the number of SQLVAR occurrences as set by the compilation unit.

b) If the <prepared statement> that is being described for input variables is a dynamic insert, update, delete, or select statement, then SQLD shall be the number of input variables specified in the prepare statement, and 0 otherwise.

c) Let D be the value of SQLD and N be the value of SQLN. If D is 0 or D is greater than the N, then no values shall be assigned to occurrences of SQLVAR. Otherwise, values shall be assigned to the first D occurrence of SQLVAR so that the first occurrence of SQLVAR contain a description of the first input variable entered in the prepared statement, and so on. The description of a input variable shall consist of expected SQLTYPE, SQLLEN, SQLSCALE, SQLNULL, and SQLNAME.

d) SQLTYPE shall be set to a code as listed above which is expected by the SQL executor for that varaiable or param (usually, the type of the associated column).

e) SQLNULL shall be set to 1 if that variable permits a NULL value and 0 otherwise.

f) The values of SQLDATA and SQLIND shall be implementor defined.

g) SQLLEN shall be set to the length of that variable and is implementor defined (usually the the length of the associated column).

h) SQLSCALE shall be set to the scale of that variable and is implementor defined (usually the scale of the associated column).

i) SQLNAME shall be set to the name of that variable or param specified in the prepare statement. If the param is unnamed, the value of SQLNAME is implementor defined.

SQLCA.

Replace section 13.1 with the fllowing section:

The SQL Communication Area (SQLCA) is the data area in which SQL Executor returns information about the results of executing each SQL statement. To include SQLCA declaration the application program should use INCLUDE statement:

```
   .....
   EXEC SQL INCLUDE SQLCA
```

The SQLCA structure for various languages is :

   1. Case:

a) For C the following elements shall appear in the communication area data structure:

```
      struct sqlca_s
          {
          char sqlcaid[8];  /* Eye-catcher set to 'SQLCA'  */
          long sqlcode;     /* Return code from execution of SQL */
                               statement. 32 bit signed int      */
          char sqlerrm[180];/* error message text with 2-byte
                                           length in front */
          char sqlerrp[32]; /* Procedure name that discovered
                                                 the error */
          long sqlerrd[6];
                           /* 0 - implementor defined        */
                           /* 1 - implementor defined        */
                           /* 2 - number of rows processed    */
                           /* 3 - estimated cost of the query */
                           /* 4 - implementor defined         */
                           /* 5 - implementor defined         */
          struct sqlcaw_s
              {
              long sqlwarn0;
              long sqlwarn1;
              long sqlwarn2;
              long sqlwarn3;

              long sqlwarn4;

              long sqlwarn5;
              long sqlwarn6;
              long sqlwarn7;
              } sqlwarn;
          };


      extern struct sqlca_s sqlca;
```

b) For COBOL the following elements shall appear in the SQL communication area (SQLCA) data structure:

```
          01  SQLCA.
              05  SQLCAID          PIC X(8).
              05  SQLCODE          PIC S9(9) COMPUTATIONAL.
              05  SQLERRM          PIC X(180).
              05  SQLERRP          PIC X(32).
              05  SQLERRD          PIC S9(9) COMPUTATIONAL
                                             OCCURS 6 TIMES.
              05  SQLWARN.
                  10  SQLWARN0 PIC S9(9).
                  10  SQLWARN1 PIC S9(9).
                  10  SQLWARN2 PIC S9(9).
                  10  SQLWARN3 PIC S9(9).
                  10  SQLWARN4 PIC S9(9).
                  10  SQLWARN5 PIC S9(9).
                  10  SQLWARN6 PIC S9(9).
                  10  SQLWARN7 PIC S9(9).
```

c) For PL1 the following elements shall appear in the SQL communication area (SQLCA) data structure:

```
           1  SQLCA
              2  SQLCAID         CHAR(8),
              2  SQLCODE         BIN FIXED(32),
              2  SQLERRM         CHAR(180) VAR,
              2  SQLERRP         CHAR(32),
              2  SQLERRD(6)      BIN FIXED(31),
              2  SQLWARN,
                 3  SQLWARN0 BIN FIXED(32),
                 3  SQLWARN1 BIN FIXED(32),
                 3  SQLWARN2 BIN FIXED(32),
                 3  SQLWARN3 BIN FIXED(32),
                 3  SQLWARN4 BIN FIXED(32),
                 3  SQLWARN5 BIN FIXED(32),
                 3  SQLWARN6 BIN FIXED(32),
                 3  SQLWARN7 BIN FIXED(32);
```

d) For PASCAL the following elements shall appear in the SQL communication area (SQLCA) data structure:

```
    SQLWARN_TYPE = RECORD
           sqlwarn0  :  integer; { 32 bit signed integer }
           sqlwarn1  :  integer; { 32 bit signed integer }
           sqlwarn2  :  integer; { 32 bit signed integer }
           sqlwarn3  :  integer; { 32 bit signed integer }
           sqlwarn4  :  integer; { 32 bit signed integer }
           sqlwarn5  :  integer; { 32 bit signed integer }
```

```
            sqlwarn6  :  integer; { 32 bit signed integer }
            sqlwarn7  :  integer; { 32 bit signed integer }
            END;

    SQLCA_TYPE = RECORD
        sqlcaid     : array[1:8] of char;
                                { Eye-catcher set to 'SQLCA'          }
        sqlcode     : integer;
                                { Return code from execution of SQL }
                                { statement. 32 bit signed integer. }
        sqlerrm     : array[1:180] of char;
                                { error message text with           }
                                { 2-byte length in front            }
        sqlerrp     : array[1:32] of char;
                                { Procedure name that discovered the}
                                { error                             }
        sqlerrd     : array[1..6] of integer;
                                { 0 - implementor defined           }
                                { 1 - implementor defined           }
                                { 2 - number of rows processed      }
                                { 3 - estimated cost of the query   }
                                { 4 - implementor defined           }
                                { 5 - implementor defined           }

        sqlwarn     : SQLWARN_TYPE;
        END;


    VAR  sqlca SQLCA_TYPE;
```

GENERAL RULES:

1. When <sqlca descriptor> SQLCA is used in INCLUDE statement, the execution of every SQL statement will return error information the SQLCA. The information returned in SQLCA shall be as follows:

   a. SQLCAID : Will be set to 'SQLCA ' when the program first uses the structure.

   b. SQLCODE : Summarizes the result of executing a SQL statement. In general, zero denotes successful execution. Code greater than zero denote normal conditions experienced while executing the statement, such as an end of scan or some specific warning conditions. End of scan (file/table) is indicaterd by a +100 in the SQLCODE. Negative codes represent various abnormal conditions, which may have been caused by either an error in user program or a system failure. User should take appropriate action under these conditions. These negative error codes and positive error code other than +100 are implementor defined.

c. SQLERRM : This buffer contains the parameter text associated with a particular SQLCODE. The first two bytes of this buffer contain the total length of the string. The parameters are separated by a character of value X'FF'. For example, if the SQLCODE error code says that a table is not found then SQLERRM will contain the table name.

d. SQLERRP : If the SQLCODE is negative, SQLERRP contains the name of the procedure (COBOL/C/PSACAL/PL1/FORTAN) that discovered the error. This buffer may also contain the line number within that procedure where appropriate.

e. SQLERRD : This is a collection of six variables that describe the current state of the system. In this variable

3. Number of rows processed, where applicable.

4. Estimated cost of a query execution. A relative value incorporating I/O, CPU, and any other related factor in estimating the cost of a query execution. This field can be used to estimate the relative performance of the prepared SQL statement.

The variables 1,2,5, and 6 are implementor defined.

f. SQLWARN : This is a collection 8 integer variables which can be used to represent one or more warnings raised by the SQL executor during query compilation and/or execution. The values are implementor defined.