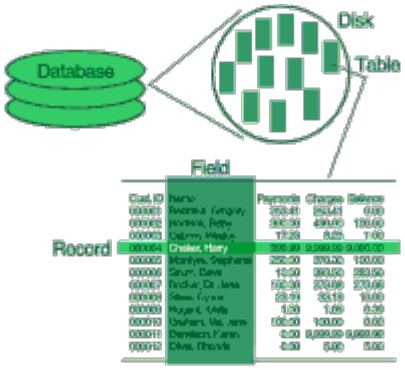
SQL Access in PRO/5

By Russ Kepler

PRO/5 provides verbs and functions to simplify access to databases external to PRO/5, and to allow SQL access to PRO/5 datafiles. The commands are designed to provide maximum access to the SQL capabilities available in the BASIS SQL Engine and in other SQL products with which PRO/5 is designed to communicate.

SQL data is conceptualized very differently than data in PRO/5 and other DBMS systems that are based on the concept of files and records. If you wish to attempt to relate the concepts you can only approach similarity. In SQL the "database" is close to the concept of splitting of data between functional programs - accounts payable, accounts receivable, etc., but may be closer to splitting data between organizations or groups within organizations.

Once a database is chosen you can access tables in the database. A TABLE in SQL is close in concept to the FILE in PRO/5 - a division of data below the database level. In SQL, however, a table doesn't have to map to a particular file and may be a group of data from multiple tables or part of another table (called a VIEW).



Inside tables are "rows", roughly analogous to records in files, and rows consist of "columns", again analogous to fields in records. The primary difference is that SQL will never allow multiple record types in a field, something unfortunately common in file/record systems.

PRO/5 data access commands map to SQL commands as follows:

PRO/5 command	SQL command
READ	SELECT
WRITE	INSERT/UPDATE
REMOVE	DELETE
create "file"	CREATE TABLE
erase "file"	DROP TABLE

DATABASE SELECTION

The SQLLIST() function returns a list of databases for which PRO/5 is configured to handle.

The SQLOPEN verb selects a database. It associates a user specified channel with a database selected from those listed in the SQLLIST() function.

SQL COMMAND EXECUTION

Once a database is selected a list of available tables may be acquired by using the SQLTABLES() function. SQL commands are executed in two steps. First an SQL command must be run through SQLPREP. SQLPREP checks the legality of the command and prepares an execution strategy. The second step is using SQLEXEC. SQLEXEC executes the command that was prepared by SQLPREP.

If the SQL command (or commands) contain replaceable arguments they may be adjusted using the SQLARG verb or, optionally, using the SQLEXEC verb.

READING DATA FROM SQL

Once the SQL command has been executed using the SQLEXEC verb the data may be returned to the programmer from the SQLFETCH() function. When all of the data from the SQL command is returned the SQLFETCH() function will return an !ERROR=2. If any other errors are reported by either the SQLPREP or SQLEXEC verbs an explanation of the error is available in the SQLERR() function. You may terminate a database connection with the SQLCLOSE verb.

As an example consider accessing the database "Chile Company", available in TAOS/*Views*. In that demonstration database there are several tables available, and the following program shows selection of data from one of them.

First, you need to attach to a database. Since you know the database in advance you can bypass the SQLLIST() function and directly attach to the database:

```
0010 SQLOPEN(1)"Chile Company"
```

Knowing the database you can also execute the command that you desire, in this case a selection of the customers with a current balance:

```
0020 SQLPREP(1)"select * from CUSTOMER where CURRENT_BAL>0"
```

You can now format a string to receive the rows of data generated by the SQL select by using SQLTMPL():

```
0030 DIM CUST$:SQLTMPL(1)
```

With everything in place to receive the data, you tell SQL that all is ready for execution:

```
0040 SQLEXEC(1)
```

Since the select command isn't ordering the data much you will likely be able to start receiving data immediately, so you execute:

```
0050 CUST$=SQLFETCH(1,ERR=90)
```

The error branch will handle the end of the select set (similar to the "end of file" condition encountered when reading data from a file), and if the error branch is not taken the variable "cust$" will receive the first/next row from the table selected by the SQL select command. You will do something with it and go fetch another:

```
0060 PRINT (7)CUST.CUST_NUM$," ",CUST.COMPANY$," ",CUST.CURRENT_BAL
0070 GOTO 50
```

The error branch should handle the expected case of the "end of file , and generate error diagnostics for the other, unexpected, conditions:

```
0080 IF ERR=2 THEN GOTO0110
```

```
0090 PRINT "error",ERR,"encountered, sql error text is:"
0100 PRINT SQLERR(1,ERR=0110); GOTO0100
0110 END
```

The SQLFETCH() function will fetch any data available as a result of the SQLPREP and SQLEXEC verbs. If no data is available as a result of the SQL command, or if all data made available as a result of the executed command has already been fetched, SQLFETCH() will return an !ERROR=2. If a command is executed that prepares a list of rows to be returned, and you prepare and execute a new command, the old list of rows is simply discarded without complaint. Additionally, you can *re-start* a query by simply doing another SQLEXEC, without requiring a new SQLPREP. This is good practice since SQLPREP can be expensive.

Inserting a Row in a Table

To insert a row in a database the following commands would be used:

```
0010 SQLOPEN(1)"Chile Company"
0020 SQLPREP(1)"insert into CUSTOMER CUST_NUM=6,FIRST_NAME='Ralph',
      LAST_NAME='Hammerschmidt'"
0030 SQLEXEC(1)
```

Modifying a Row in a Table

SQLPREP will insert into the CUSTOMERS table a new row with the information given in the proper columns, and all unspecified columns containing default data for that column. If you wished to modify the data in a particular customers data you might:

```
0010 SQLOPEN(1)"Chile Company"
0020 SQLPREP(1)"update CUSTOMER set CREDIT_CODE='01' where CUST_NUM=10"
0030 SQLEXEC(1)
```

Removing a Row in a Table

And to remove a row:

```
0010 SQLOPEN(1)"Chile Company"
0020 SQLPREP(1)"delete from CUSTOMER where CUST_NUM=10"
0030 SQLEXEC(1)
```

Multiple SQLPREP COMMANDS

The above examples create a new connection and compile a new command for each operation desired. In practice it is generally easier to maintain multiple connections to the database, each with previously compiled commands with settable arguments, to select the row on which to operate. It would be more effective for many of the above operations to perform something like the following:

```
0010 DATABASE$="Chile Company"
0020 GETREC=SQLUNT; SQLOPEN(GETREC)DATABASE$
0021 SQLPREP(GETREC)"select * from CUSTOMER where CUST_NUM=?"
0030 DELREC=SQLUNT; SQLOPEN(DELREC)DATABASE$
0031 SQLPREP(DELREC)"delete from CUSTOMER where CUST_NUM=?"
0040 DIM CUST$:SQLTMPL(GETREC)
0050 INPUT "customer number: ",CUST:("end"=1000,9999)
0060 SQLEXEC (GETREC)CUST
0070 CUST$=SQLFETCH(GETREC,ERR=0200)
0080 PRINT CUST.CUST_NUM," ",CUST.COMPANY$
0090 INPUT (0,ERR=0090)"delete? (y/n) ",ANSWER$:("Y"=100,"y"=100,"n"=50,"N"=50)
0100 SQLEXEC(DELREC)CUST
0110 PRINT CUST.CUST_NUM," deleted"
0120 GOTO 0050
0200 PRINT "record not found"; GOTO 0050
```