```
process AIRPLANE
     call TOWER giving GATE yielding RUNWAY
     work TAXI.TIME (GATE, RUNWAY) minutes
     request 1 RUNWAY
     work TAKEOFF.TIME (AIRPLANE) minutes
     relinquish 1 RUNWAY
end " process AIRPLANE

process AIRPLANE
     call TOWER giving GATE yielding RUNWAY
     work TAXI.TIME (GATE, RUNWAY) minutes
     request 1 RUNWAY
     work TAKEOFF.TI
     relinquish 1
end " process AI
```

Database Connectivity User's Manual

CACI

For product information or technical support contact:

CACI Products Company
1011 Camino Del Rio South, Suite 230
San Diego, CA 92108
Phone: (619) 542-5228
Fax: (619) 692-1013

Table of Contents

Chapter 1 Introduction to SDBC

SIMSCRIPT II.5® Database Connectivity (SDBC) is a library of functions and routines that enables SIMSCRIPT II.5 programs to access databases. SDBC makes it possible for SIMSCRIPT II.5 programs to create tables in relational databases; to insert, modify, and delete the rows of database tables; and to perform database queries.

To use SDBC, it is necessary to have installed a database management system (DBMS). There are many to choose from, including Microsoft Access, Microsoft SQL Server, IBM DB2®, IBM Informix, and Oracle®. SDBC provides a common interface to all of these.

SDBC is patterned after and utilizes Microsoft's Open Database Connectivity (ODBC). SDBC works with any DBMS having an ODBC 3.0 driver. SIMSCRIPT II.5 programs call SDBC functions and routines, which in turn call ODBC functions that communicate with the DBMS through this driver.

This manual assumes that the reader has a working knowledge of the SIMSCRIPT II.5 programming language and is familiar with relational database concepts, including Structured Query Language (SQL).

1.1 Setting Up a Database

The first step is to create a database. Each DBMS provides its own mechanism for creating a database. For example, a database is created in Microsoft Access by selecting New from the File menu. Consult the DBMS documentation to learn how to create a database.

The second step is to define an *ODBC data source*, which associates an *ODBC data source name* with the database and specifies the ODBC 3.0 driver to use. In Microsoft Windows®, this is accomplished by running the ODBC Data Source Administrator program which can be found in the Control Panel.

1.2 Declaring the SDBC Functions and Routines

The SDBC functions and routines must be declared in the Preamble of the SIMSCRIPT II.5 program. A text file containing the required declarations is provided as part of the SDBC installation. Copy the contents of this file into the Preamble:

```
preamble

...

''SDBC Functions and Routines
define DB.AUTOCOMMIT.R as a          routine  given 1 argument
define DB.COMMIT.R     as a          routine  given 0 arguments
define DB.CONNECT.R    as a          routine  given 3 arguments
define DB.DISCONNECT.R as a          routine  given 0 arguments
define DB.EXISTS.F     as an integer function given 1 argument
define DB.FETCH.F      as an integer function given 0 arguments
define DB.GETINT.F     as an integer function given 1 argument
define DB.GETREAL.F    as a  double  function given 1 argument
define DB.GETTEXT.F    as a  text    function given 1 argument
define DB.NULL.F       as an integer function given 1 argument
define DB.QUERY.R      as a          routine  given 1 argument
define DB.ROLLBACK.R   as a          routine  given 0 arguments
define DB.SETINT.R     as a          routine  given 2 arguments
define DB.SETREAL.R    as a          routine  given 2 arguments
define DB.SETTEXT.R    as a          routine  given 2 arguments
define DB.UPDATE.F     as an integer function given 1 argument

...

end
```

1.3 Connecting to a Database

Before any operations can be performed on a database, the SIMSCRIPT II.5 program must first connect to the database. This is accomplished by calling **DB.CONNECT.R**:

```
call DB.CONNECT.R(DSNAME, USERNAME, PASSWORD)
```

DSNAME is a text value specifying the ODBC data source name associated with the database. If the database has been set up for secure access, then text values **USERNAME** and **PASSWORD** must provide a valid user name and password for this database. If no security has been established for this database, then any user name and password may be given to this routine. (Consult the DBMS documentation for information on how to secure a database.)

Example:

```
define USER, PWD as text variables

write as "Enter your database user name:", /
read USER
write as "Enter your database password:", /
read PWD

call DB.CONNECT.R("TESTDB1", USER, PWD)
```

When finished with the database, the program calls **DB.DISCONNECT.R** to disconnect:

```
call DB.DISCONNECT.R
```

It is not possible to connect to more than one database at a time. However, after disconnecting from one database, the program may connect to a second database (or reconnect to the first database). If the program calls **DB.CONNECT.R** while already connected to a database, an implicit disconnection occurs before the new connection is attempted. A program that terminates while connected to a database is implicitly disconnected.

1.4 Interpreting Run-time Errors

SIMSCRIPT II.5 run-time error #2400 is generated for every SDBC-related error. For example, if an invalid ODBC data source name is passed to **DB.CONNECT.R**, the following error message may be produced:

```
RUN-TIME ERROR #2400: [IM002][0][Microsoft][ODBC Driver
Manager] Data source name not found and no default driver
specified
```

The error message may have come from the DBMS and may be too long to fit in the SIMSCRIPT II.5 SimDebug display. In this case, the message is truncated:

```
RUN-TIME ERROR #2400: [23000][-1605][Microsoft][ODBC Microsoft
Access Driver] The changes you requested to the table were
not successful...
```

Refer to the file named **SDBC.log** in the current working directory for the full message:

```
SDBC Run-time Error, Thu Jan 10 14:03:18 2002
[23000][-1605][Microsoft][ODBC Microsoft Access Driver] The
changes you requested to the table were not successful because
they would create duplicate values in the index, primary key,
```

```
or relationship.  Change the data in the field or fields that
contain duplicate data, remove the index, or redefine the index
to permit duplicate entries and try again.
```

As illustrated by these examples, an SDBC run-time error message may contain special information in brackets. The first value in brackets (e.g., **23000**) is an *SQLSTATE* code; see Appendix C for a list of these codes and their meanings. The second value in brackets (e.g., **-1605**) is an error code specific to the DBMS; see the DBMS documentation for details. The other bracketed information identifies the vendor, ODBC component, and DBMS from which the error message came.

Chapter 2 SQL Updates

After connecting to a database via **DB.CONNECT.R**, the SIMSCRIPT II.5 program may pass SQL statements one at a time to the DBMS for processing. An SQL statement that modifies the database is executed by **DB.UPDATE.F**, which is discussed in this section. An SQL statement that queries the database, without modifying it, is processed by **DB.QUERY.R**, which is covered in Section 3. The following is an example of a **DB.UPDATE.F** call:

```
NUMROWS = DB.UPDATE.F(COMMAND)
```

COMMAND is a text value giving the SQL statement to be processed. This function returns after the given statement has been executed by the DBMS on the connected database. The integer return value indicates the number of rows affected by the execution of this statement, if applicable.

2.1 Creating Tables

Passing an `SQL CREATE TABLE` statement to `DB.UPDATE.F` creates a database table A **CREATE TABLE** statement names the table and its columns, and specifies the data type for each column. The following SQL data types are supported by almost every DBMS:

SMALLINT	a signed 16-bit integer
INTEGER	a signed 32-bit integer
REAL	a single-precision floating-point number
DOUBLE	a double-precision floating-point number
CHAR(*n*)	a fixed-length character string of length *n*
VARCHAR(*n*)	a variable-length character string having a maximum length of *n*

A DBMS may permit a variety of synonyms for these data types, such as **SHORT** for **SMALLINT**; **INT** or **LONG** for **INTEGER**; **SINGLE** for **REAL**; **DOUBLE PRECISION** for **DOUBLE**; **CHARACTER** for **CHAR**; and **CHAR VARYING** or **CHARACTER VARYING** for **VARCHAR**. A DBMS may also support a variety of other data types, such as **BOOLEAN**, **BYTE**, **COUNTER**, **DECIMAL**, **DATE**, and **TIME**. Also, a DBMS may require that long character strings be stored as a special data type called **TEXT**, **LONGTEXT**, or **LONG VARCHAR**. Consult the DBMS documentation for details.

The following SQL statement creates a table named **RESULT**. Each row in this table will record the result of one simulation run.

```
CREATE TABLE RESULT
(RUNID   INTEGER NOT NULL PRIMARY KEY,
 MAXQLEN INTEGER,
 AVGQLEN REAL,
 COMMENT VARCHAR(80))
```

This table has four columns: **RUNID**, **MAXQLEN**, **AVGQLEN**, and **COMMENT**. **RUNID** holds an integer ID that uniquely identifies the simulation run; therefore, this column has been designated as the *primary key* for the table. **MAXQLEN** contains an integer value giving the maximum queue length observed during the run. **AVGQLEN** holds a single-precision floating-point value giving the average queue length observed during the run. **COMMENT** provides space for a text comment, up to 80 characters in length. Each column may be undefined and assigned a null value, except **RUNID** which has been designated as **NOT NULL** and must always contain a non-null value.

To create this table, the **CREATE TABLE** statement is passed as a text value to **DB.UPDATE.F**. Since the text value is rather long, we use **CONCAT.F** to construct it:

```
define CMD  as a text variable
define ROWS as an integer variable

CMD = CONCAT.F(
"CREATE TABLE RESULT ",
"(RUNID   INTEGER NOT NULL PRIMARY KEY,",
" MAXQLEN INTEGER,",
" AVGQLEN REAL,",
" COMMENT VARCHAR(80))")

ROWS = DB.UPDATE.F(CMD)
```

Upon return from **DB.UPDATE.F**, a table has been created with the specified name and columns, containing no rows. The return value in **ROWS** is undefined and should be ignored.

To destroy this table, pass a **DROP TABLE** statement to **DB.UPDATE.F**:

```
ROWS = DB.UPDATE.F("DROP TABLE RESULT")
```

Refer to Appendix B in this manual, and the DBMS documentation, for a specification of the syntax of the **CREATE TABLE** and **DROP TABLE** statements.

CREATE TABLE and **DROP TABLE** are examples of SQL Data Definition Language (DDL) statements. A DBMS may support many other types of DDL statements, including **ALTER TABLE**, **CREATE/DROP VIEW**, **CREATE/DROP INDEX**,

CREATE/ALTER/DROP DOMAIN, **CREATE/DROP ASSERTION**, and **GRANT/REVOKE**. See the DBMS documentation for details. Any DDL statement may be passed to **DB.UPDATE.F** for execution. For all DDL statements, the return value from **DB.UPDATE.F** is undefined and should be ignored.

SDBC supplies a function named **DB.EXISTS.F** to determine whether a table exists. This function takes a table name as its only argument and returns 1 if the table exists or 0 if the table does not exist. To avoid a run-time error for attempting to create a table that already exists, call this function to verify that the table does not exist before creating it:

```
if DB.EXISTS.F("RESULT") = 0 ''the table does not exist
   ''create the table
   ...
always
```

Likewise, to avoid a run-time error for attempting to drop a table that does not exist, call **DB.EXISTS.F** to verify that the table exists before dropping it:

```
if DB.EXISTS.F("RESULT") = 1 ''the table exists
   ''drop the table
   ...
always
```

Please note that SIMSCRIPT II.5 programs can access tables that have been created by other means, such as by an interactive SQL command processor supplied by the DBMS; and DBMS tools can access tables created by SIMSCRIPT II.5 programs.

2.2 Inserting Rows

To insert a row into a table, an SQL **INSERT** statement is passed to **DB.UPDATE.F**. The following code inserts a row into the **RESULT** table, setting **RUNID** to 101, **MAXQLEN** to 12, **AVGQLEN** to 2.75, and **COMMENT** to "First test run in December":

```
define CMD  as a text variable
define ROWS as an integer variable

CMD = CONCAT.F(
"INSERT INTO RESULT",
" VALUES (101, 12, 2.75, 'First test run in December')")

ROWS = DB.UPDATE.F(CMD)
```

Upon return from **DB.UPDATE.F**, the specified row has been inserted into the table. The return value in **ROWS** is 1, indicating that one row has been inserted. Note that

text literals in SQL are delimited by single quotes, not double quotes as in SIMSCRIPT II.5.

When one or more columns are undefined, a variant of the SQL **INSERT** statement may be used that specifies only the defined columns of the new row. The following code inserts a row into the **RESULT** table, setting **RUNID** to 200 and **COMMENT** to "Demo". To **MAXQLEN** and **AVGQLEN**, which are omitted, null values are assigned implicitly. (If the DBMS supports default values, the omitted columns receive their default values, which may be non-null.)

```
ROWS = DB.UPDATE.F(
"INSERT INTO RESULT (RUNID, COMMENT) VALUES (200, 'Demo')")
```

Or null values may be specified explicitly for the undefined columns:

```
ROWS = DB.UPDATE.F(
"INSERT INTO RESULT VALUES (200, NULL, NULL, 'Demo')")
```

Another variant of the **INSERT** statement specifies a query and inserts each row returned by the query.

The **INSERT** statement is an SQL Data Manipulation Language (DML) statement. Refer to Appendix B in this manual, and the DBMS documentation, for a specification of its syntax.

2.3 Modifying Rows

To modify the value of one or more columns in one or more rows, an SQL **UPDATE** statement is passed to **DB.UPDATE.F**. The following code changes the values of **MAXQLEN** to 10 and **AVGQLEN** to 2.25 in the row that has **RUNID** equal to 101:

```
define CMD  as a text variable
define ROWS as an integer variable

CMD = CONCAT.F(
"UPDATE RESULT",
" SET MAXQLEN = 10, AVGQLEN = 2.25",
" WHERE RUNID = 101")

ROWS = DB.UPDATE.F(CMD)
```

Upon return from **DB.UPDATE.F**, the requested modification has been performed. The return value in **ROWS** indicates the number of rows modified. Presumably **ROWS=1** in our example; however, it could be zero if there does not exist a row having **RUNID=101**, or greater than one if more than one row has **RUNID=101**.

If no **WHERE** clause is specified in the **UPDATE** statement, then the modification is applied to every row in the table. For example, the following code adds 1000 to every **RUNID**:

```
ROWS = DB.UPDATE.F("UPDATE RESULT SET RUNID = RUNID + 1000")
```

In this case, the return value in **ROWS** equals the number of rows in the table since every row was modified.

The following example sets the **COMMENT** field to null for every **RUNID** greater than 1200:

```
ROWS = DB.UPDATE.F(
"UPDATE RESULT SET COMMENT = NULL WHERE RUNID > 1200")
```

The **WHERE** clause may specify any conditional expression allowed in SQL. Expressions are discussed in Section 3.2.

The **UPDATE** statement is an SQL Data Manipulation Language (DML) statement. Refer to Appendix B in this manual, and the DBMS documentation, for a specification of its syntax.

2.4 Deleting Rows

To delete one or more rows, an SQL **DELETE** statement is passed to **DB.UPDATE.F**. The following code deletes the row that has **RUNID** equal to 101:

```
define ROWS as an integer variable

ROWS = DB.UPDATE.F("DELETE FROM RESULT WHERE RUNID = 101")
```

Upon return from **DB.UPDATE.F**, the requested deletion has been performed. The return value in **ROWS** indicates the number of rows deleted. Presumably **ROWS=1** in our example; however, it could be zero if there did not exist a row having **RUNID=101**, or greater than one if more than one row had **RUNID=101**.

If no **WHERE** clause is specified in the **DELETE** statement, then every row in the table is deleted:

```
ROWS = DB.UPDATE.F("DELETE FROM RESULT")
```

In this case, the return value in **ROWS** equals the number of rows that were in the table before they were all deleted.

The following example deletes all rows having a **RUNID** greater than or equal to 1000 and less than 2000:

```
ROWS = DB.UPDATE.F(
"DELETE FROM RESULT WHERE RUNID >= 1000 AND RUNID < 2000")
```

The **WHERE** clause may specify any conditional expression allowed in SQL. Expressions are discussed in Section 3.2.

The **DELETE** statement is an SQL Data Manipulation Language (DML) statement. Refer to Appendix B in this manual, and the DBMS documentation, for a specification of its syntax.

Chapter 3 SQL Queries

After connecting to a database via **DB.CONNECT.R**, the SIMSCRIPT II.5 program may submit queries to the DBMS for processing. The SQL **SELECT** statement is used to query the database. After executing a query, the rows returned by the query are retrieved by the program.

3.1 Querying the Database

Any **SELECT** statement may be passed as a text value to **DB.QUERY.R** for execution:

```
call DB.QUERY.R("SELECT ...")
```

The rows returned by the query are then fetched one at a time by calling **DB.FETCH.F** for each row. This function returns 1 if it has successfully fetched the next row and returns 0 when there are no more rows.

After fetching a row, the values of the columns in the row are obtained by calling **DB.GETINT.F** for each integer column, **DB.GETREAL.F** for each floating-point column, and **DB.GETTEXT.F** for each character-string column.

The following query returns all rows in the table named **RESULT**:

```
SELECT * FROM RESULT
```

The asterisk is shorthand for listing all of the columns of the table, in the order in which they were defined in the **CREATE TABLE** statement. So the above query is equivalent to:

```
SELECT RUNID, MAXQLEN, AVGQLEN, COMMENT FROM RESULT
```

The following code is typical of SDBC query processing. First, the query is executed, and then a **while** loop fetches the rows returned by the query. For each row, the column values are obtained and then some code is executed that uses these values. In this example, program variables are given the same names as columns, but this is done only for clarity and is not a requirement. However, the modes of these variables should match the SQL data types of the columns.

```
define RUNID, MAXQLEN as integer variables
define AVGQLEN as a real variable
define COMMENT as a text variable

''execute the query
call DB.QUERY.R("SELECT * FROM RESULT")

''fetch the rows
while DB.FETCH.F = 1
do
   ''a row has been fetched;
   ''now obtain the value of each column
   RUNID   = DB.GETINT.F(1)
   MAXQLEN = DB.GETINT.F(2)
   AVGQLEN = DB.GETREAL.F(3)
   COMMENT = DB.GETTEXT.F(4)
   ''do some processing using these values
   ...
loop
```

One row is processed in each iteration of the **while** loop. **DB.FETCH.F** returns 0 when there are no more rows, which terminates the loop.

A *column number* is the ordinal position of a column within a row. For this query, **RUNID** is column number 1, **MAXQLEN** is column number 2, **AVGQLEN** is column number 3, and **COMMENT** is column number 4. **DB.GETINT.F**, **DB.GETREAL.F**, and **DB.GETTEXT.F** return the value of the column identified by the given column number. Specifying a column number less than 1 or greater than the number of columns produces a run-time error.

When the value of a column is null, **DB.GETINT.F** returns 0, **DB.GETREAL.F** returns 0.0, and **DB.GETTEXT.F** returns the zero-length string (""). However, these values can also be returned for non-null columns. Therefore, SDBC supplies a function named **DB.NULL.F** to determine whether a column contains a null value. This function accepts a column number as its only argument and returns 1 if the value of the column is non-null and returns 0 if the value is null. For example:

```
if DB.NULL.F(3) = 0 ''column number 3 is null
   ...
always

if DB.NULL.F(2) = 1 ''column number 2 is non-null
   ...
always
```

Note that **DB.GETINT.F**, **DB.GETREAL.F**, **DB.GETTEXT.F**, and **DB.NULL.F** refer only to the most recently fetched row. It is not possible to access any other row. Likewise, **DB.FETCH.F** fetches only rows returned by the most recent query. It is not possible to fetch rows returned by a prior query.

It is not necessary to retrieve all column values if the program only needs some. It is also not required to fetch all of the rows returned by the query; that is, the program may terminate the **while** loop early, before all rows have been retrieved.

After **DB.FETCH.F** has returned 0, which indicates there are no more rows to be fetched, calling **DB.FETCH.F** again, without first executing a new query, produces a run-time error.

If **DB.GETINT.F** is called to retrieve a floating-point column value, it returns the value rounded to the nearest integer. If **DB.GETREAL.F** retrieves an integer column value, it returns the value as a real number. If **DB.GETINT.F** or **DB.GETREAL.F** retrieves a character-string column value, it attempts to convert the value to a number. If **DB.GETTEXT.F** retrieves an integer or floating-point column value, it converts the value to text.

3.2 Specifying SQL Expressions

Like SIMSCRIPT II.5 names, SQL names are case-insensitive and may consist of any combination of letters and digits; however, an SQL name must begin with a letter. An SQL name may not contain periods unless it is a qualified name, such as a column name qualified by a table name (e.g., **RESULT.RUNID**). An SQL name may contain underscores (e.g., **FIRST_NAME**). See the DBMS documentation for a list of reserved SQL key words (e.g., **SELECT**, **FROM**, **INSERT**), which may not be used to name a table or column.

Numeric constants in SQL and SIMSCRIPT II.5 are specified in the same way, except that SQL permits scientific notation in constants (e.g., **3.87E-4**). Text literals in SQL are delimited by 'single quotes', rather than "double quotes" as in SIMSCRIPT II.5: **'This is an SQL text literal'**, **'Embedded quotes aren''t a problem'**, the zero-length string looks like this **''**.

SQL and SIMSCRIPT II.5 share the following operators:

```
+  -  *  /  AND  OR  =  <>  <  >  <=  >=
```

In SQL, **NOT** may be used for logical negation. SQL does not have an exponentiation operator (******) and does not support any of the English abbreviations (e.g., **EQ**, **LT**) or phrases (e.g., **EQUALS**, **LESS THAN**) allowed in SIMSCRIPT II.5. SQL permits the following expressions to test for nulls: **X IS NULL**, **X IS NOT NULL**.

SIMSCRIPT II.5's concise **0 < X < 100** must be expressed in SQL as **0 < X AND X < 100**. However, **0 <= X <= 100** may be expressed in SQL as **X BETWEEN 0 AND 100**. Its negation, expressed as **0 <= X <= 100 IS FALSE** in

SIMSCRIPT II.5, is expressed in SQL as **`NOT (X BETWEEN 0 AND 100)`** or simply, **`X NOT BETWEEN 0 AND 100`**.

SQL's **IN** operator provides convenient shorthand for testing whether a column value belongs to a list of values:

```
X IN (20, 21, 26, 31, 32)
CITY NOT IN ('Detroit', 'Chicago', 'Cincinnati')
```

In SQL, a **SELECT** statement known as a *subquery* may appear within an expression. Given a subquery as an operand, the **IN** operator returns true if a given value is returned by the subquery, and the **EXISTS** operator returns true if the subquery returns at least one row. A *scalar subquery* returns the value of a single column in a single row.

Full treatment of SQL expressions is beyond the scope of this manual. Please refer to a book on SQL and the DBMS documentation for more information.

3.3 Selecting Rows

The **WHERE** clause in a **SELECT** statement specifies an SQL conditional expression. Only rows for which the expression evaluates to true are returned by the query. The following query returns the **RUNID**, **AVGQLEN**, and **COMMENT** columns of each row in **RESULT** that has a **RUNID** between 2000 and 2999 and an **AVGQLEN** greater than or equal to 2.0:

```
SELECT RUNID, AVGQLEN, COMMENT
 FROM RESULT
 WHERE RUNID BETWEEN 2000 AND 2999
   AND AVGQLEN >= 2.0
```

The rows returned by a query are unordered unless an **ORDER BY** clause is specified. To sort the rows by descending **AVGQLEN** and then by ascending **RUNID** for rows having the same **AVGQLEN**, the following clause is appended to the **SELECT** statement:

```
ORDER BY AVGQLEN DESC, RUNID ASC
```

Because ascending is the default, the **ASC** key word may be omitted. Column numbers may be specified in place of the column names. For this query, **RUNID** is column number 1, **AVGQLEN** is column number 2, and **COMMENT** is column number 3; therefore, the following **ORDER BY** clause is equivalent to the one above:

```
ORDER BY 2 DESC, 1
```

The following code executes the above query and prints the five longest average queue lengths. If the query returns fewer than five rows, then all of the rows will be

fetched and printed. If the query returns more than five rows, then only the first five will be fetched and printed.

```
define CMD as a text variable
define I   as an integer variable

''construct the query
CMD = CONCAT.F(
"SELECT RUNID, AVGQLEN, COMMENT FROM RESULT",
" WHERE RUNID BETWEEN 2000 AND 2999 AND AVGQLEN >= 2.0",
" ORDER BY 2 DESC, 1")

''execute the query
call DB.QUERY.R(CMD)

''print column headings
print 1 line as follows
   AVGQLEN   RUNID   COMMENT

''fetch and print the first five rows
for I = 1 to 5 while DB.FETCH.F = 1
   print 1 line with DB.GETREAL.F(2), DB.GETINT.F(1),
   DB.GETTEXT.F(3) as follows
      *.**       *   ******************************
```

The output might look like this:

```
AVGQLEN   RUNID   COMMENT
 118.38    2391   Extremely slow server
  41.91    2877
  17.00    2017   Test M7
  17.00    2018   Test M8
  14.96    2450   Tried a Weibull distribution
```

In this example, the row with **RUNID=2877** has a null **COMMENT**, which is returned by **DB.GETTEXT.F** as a zero-length string ("") and gets printed as blanks.

SQL provides the following *aggregate functions*:

COUNT(*)	returns the number of rows
AVG(*column***)**	returns the average of the values in *column*
MAX(*column***)**	returns the largest value in *column*
MIN(*column***)**	returns the smallest value in *column*
SUM(*column***)**	returns the sum of the values in *column*

The following code uses aggregate functions to report the number of rows in **RESULT** and the minimum, maximum, and average value of **AVGQLEN**:

```
''execute the query
call DB.QUERY.R(CONCAT.F(
"SELECT COUNT(*), MIN(AVGQLEN), MAX(AVGQLEN), AVG(AVGQLEN)",
" FROM RESULT"))

if DB.FETCH.F = 1 ''fetched the only row
   write DB.GETINT.F(1), DB.GETREAL.F(2), DB.GETREAL.F(3),
   DB.GETREAL.F(4) as "In ", I 4,
   " simulation runs, the average queue length", /,
   "ranged from ", D(4,2), " to ", D(6,2),
   " with an average of ", D(5,2), ".", /
always
```

The output might look like this:

```
In 3236 simulation runs, the average queue length
ranged from 0.15 to 172.81 with an average of  8.39.
```

Aggregate functions are commonly applied to groups of rows specified in **GROUP BY** and **HAVING** clauses of a **SELECT** statement.

The **SELECT** statement is an SQL Data Manipulation Language (DML) statement. Refer to Appendix B in this manual, and the DBMS documentation, for a specification of its syntax.

3.4 Joining Tables

One of the most important database operations is the ability to join two or more tables. This section illustrates a query that joins two tables.

In the **RESULT** table, each row records the result of one simulation run and each run is identified by a unique **RUNID**. Suppose there exists a second table, named **DETAIL**, with the following definition:

```
CREATE TABLE DETAIL
(RUNID      INTEGER NOT NULL,
 START_TIME REAL    NOT NULL,
 END_TIME   REAL    NOT NULL,
 QLEN       INTEGER NOT NULL)
```

A row in **DETAIL** indicates there was a constant queue length (**QLEN**) from simulation time **START_TIME** to **END_TIME** in the simulation run identified by **RUNID**.

The **DETAIL** and **RESULT** tables have a many-to-one relationship: for each row in **RESULT**, there are many rows in **DETAIL**. In SIMSCRIPT II.5 terminology, this relationship may be described in terms of entities and sets: each **RESULT** entity owns a set of **DETAIL** entities.

With this detailed information, a simulation run can be analyzed in greater depth. The following code calculates and displays the total simulation time for each queue length for run #2300:

```
define JOIN as a text variable
define MAXQLEN, QLEN as integer variables
define DURATION as a 1-dimensional real array

''construct a query that joins tables RESULT and DETAIL;
''since RUNID names a column in both tables, it must be
''qualified by the table name
JOIN = CONCAT.F(
"SELECT RESULT.RUNID, AVGQLEN, MAXQLEN,",
"       END_TIME - START_TIME, QLEN",
" FROM RESULT, DETAIL",               ''the tables to join
" WHERE RESULT.RUNID = DETAIL.RUNID", ''how to join them
"   AND RESULT.RUNID = 2300")

''execute the query
call DB.QUERY.R(JOIN)

if DB.FETCH.F = 0 ''the query returned no rows
   write as "There is no record of this simulation run", /
else ''fetched the first row

   write DB.GETINT.F(1) as "Run #", I 4, /
   write as "Average queue length: "
   if DB.NULL.F(2) = 1 ''AVGQLEN is non-null
      write DB.GETREAL.F(2) as D(5,2), /
   else ''AVGQLEN is null
      write as "undefined", /
   always

   if DB.NULL.F(3) = 0 ''MAXQLEN is null
      write as "Maximum queue length: undefined", /
   else ''MAXQLEN is non-null

      MAXQLEN = DB.GETINT.F(3)

      ''reserve an array with one element for each possible
      ''queue length; queue length ranges from 0 to MAXQLEN so
      ''(MAXQLEN+1) elements are needed; the duration for queue
      ''length I is summed in element (I+1)
      reserve DURATION(*) as MAXQLEN + 1
      add DB.GETREAL.F(4) to DURATION(DB.GETINT.F(5) + 1)
      while DB.FETCH.F = 1 ''fetched another row
         add DB.GETREAL.F(4) to DURATION(DB.GETINT.F(5) + 1)

      ''display the distribution of queue lengths
      write as /, "QLEN  DURATION", /
      for QLEN = 0 to MAXQLEN
         write QLEN, DURATION(QLEN + 1) as I 4, "  ", D(8,2), /
```

```
          release DURATION(*)

       always

    always
```

The output might look like this:

```
Run #2300
Average queue length:  3.28

QLEN  DURATION
   0    157.41
   1    281.31
   2    479.98
   3    394.01
   4    317.84
   5    231.09
   6    141.33
   7     89.77
   8     46.62
   9     28.16
  10     12.79
  11      6.13
  12      2.01
  13      0.89
  14      0.32
  15      0.13
```

Chapter 4 SQL Parameters

The query in Section 3.4 retrieves the data for run #2300. To process the same query on a different run, the value **2300** needs to be changed in the query. Rather than modify the query string for every new run ID, an SQL parameter, in the form of a question mark (**?**), may be specified in the query as a placeholder for the run ID. The value of this parameter is set prior to the execution of the query. For example:

```
define JOIN  as a text variable
define RUNID as an integer variable

''construct a query that uses an SQL parameter
JOIN = CONCAT.F(
"SELECT RESULT.RUNID, AVGQLEN, MAXQLEN,",
"       END_TIME - START_TIME, QLEN",
" FROM RESULT, DETAIL",
" WHERE RESULT.RUNID = DETAIL.RUNID",
"   AND RESULT.RUNID = ?")

''read the run ID
write as "Enter Run #:", /
read RUNID

''set the parameter value
call DB.SETINT.R(1, RUNID)

''execute the query using the parameter value
call DB.QUERY.R(JOIN)
```

The routine **DB.SETINT.R** sets the value of the parameter to the value of the **RUNID** variable. This value is used in place of the question mark when the query is executed by **DB.QUERY.R**. **DB.SETINT.R** is used to set an integer parameter; **DB.SETREAL.R** sets a floating-point parameter; and **DB.SETTEXT.R** sets a character-string parameter.

The following example from Section 2.2 illustrates how to insert a row into the **RESULT** table:

```
define CMD  as a text variable
define ROWS as an integer variable

CMD = CONCAT.F(
"INSERT INTO RESULT",
" VALUES (101, 12, 2.75, 'First test run in December')")

ROWS = DB.UPDATE.F(CMD)
```

If the column values for the new row are stored in program variables, then it takes some effort to construct this query. However, using SQL parameters, the task becomes easier:

```
define ROWS, RUNID, MAXQLEN as integer variables
define AVGQLEN as a real variable
define COMMENT as a text variable

''set the value of variables RUNID, MAXQLEN, AVGQLEN,
''and COMMENT to the column values of a new row
...

''set four parameter values
call DB.SETINT.R(1, RUNID)
call DB.SETINT.R(2, MAXQLEN)
call DB.SETREAL.R(3, AVGQLEN)
call DB.SETTEXT.R(4, COMMENT)

''insert the new row
ROWS = DB.UPDATE.F("INSERT INTO RESULT VALUES (?, ?, ?, ?)")
```

A *parameter number* is the ordinal position of a parameter (i.e., question mark) within an SQL statement. In this example, **DB.SETINT.R** sets parameter numbers 1 and 2 to integer values; **DB.SETREAL.R** sets parameter number 3 to a floating-point value; and **DB.SETTEXT.R** sets parameter number 4 to a text value. Failing to set a parameter before executing the SQL statement produces a run-time error. Specifying a parameter number less than 1 is also an error. Specifying a parameter number greater than the number of question marks is not an error; the extra parameter is simply ignored. Multiple parameters may be set in any order. It is not possible to set a parameter to a null value.

After the SQL statement has been processed by **DB.QUERY.R** or **DB.UPDATE.F**, all parameter values become undefined and must be set again before the next SQL statement with parameters is executed.

Chapter 5 Database Transactions

A *database transaction* is an atomic sequence of modifications to a database in which all or none of the modifications are made permanent. If the transaction is *committed*, then all of the changes are made permanent. If the transaction is *rolled back*, then all modifications made during the transaction are undone and the database is returned to the state it was in before the transaction was started.

A database may be shared and accessed concurrently by multiple users and executing programs. Transactions prevent them from seeing one another's uncommitted changes to the database, i.e., their work in progress. In addition, transactions enable the DBMS to restore a database to a known state following a system or program failure.

When Auto-Commit is ON, each SQL statement executed by **DB.UPDATE.F** is its own transaction. That is, either the statement completes successfully and all changes made by the statement are made permanent (the transaction is committed), or the statement fails and all changes made by the statement are undone (the transaction is rolled back). Auto-Commit is ON by default.

To execute two or more SQL statements atomically within a single transaction, it is necessary to turn Auto-Commit OFF. This is accomplished by passing zero to **DB.AUTOCOMMIT.R**:

```
call DB.AUTOCOMMIT.R(0)
```

With Auto-Commit OFF, all executed SQL statements are part of the same transaction, which is terminated by calling **DB.COMMIT.R** or **DB.ROLLBACK.R**. To save all changes made to the database during the transaction:

```
call DB.COMMIT.R
```

To undo all changes made to the database during the transaction:

```
call DB.ROLLBACK.R
```

After terminating a transaction, a new transaction is begun implicitly.

The following code atomically deletes all rows associated with a given simulation run from tables **RESULT** and **DETAIL**:

```
define RUNID, DELETED as integer variables

''turn Auto-Commit OFF
call DB.AUTOCOMMIT.R(0)

''read the run ID
write as "Enter # of Run to Delete:", /
read RUNID

''delete the RESULT row
call DB.SETINT.R(1, RUNID)
DELETED = DB.UPDATE.F("DELETE FROM RESULT WHERE RUNID = ?")

''delete all DETAIL rows
call DB.SETINT.R(1, RUNID)
add DB.UPDATE.F("DELETE FROM DETAIL WHERE RUNID = ?")
to DELETED

''end the current transaction,
''making all of the deletions permanent
call DB.COMMIT.R

if DELETED = 0
   write RUNID as "There are no rows to delete for Run #", I 4,
   ".", /
else
   write DELETED, RUNID as "All ", I 4, " rows for Run #", I 4,
   " have been deleted.", /
always
```

The output might look like this:

```
All  387 rows for Run #1542 have been deleted.
```

Once Auto-Commit has been turned OFF, it remains OFF until it is explicitly turned ON by passing a non-zero value to **DB.AUTOCOMMIT.R**:

```
call DB.AUTOCOMMIT.R(1)
```

Turning Auto-Commit ON implicitly terminates and commits the current transaction. When Auto-Commit is ON, calling **DB.COMMIT.R** or **DB.ROLLBACK.R** has no effect.

Any transaction that is ongoing when a SIMSCRIPT II.5 program terminates is automatically rolled back by the DBMS.

Chapter 6 Example Program: Bank Simulation

This section presents a complete SIMSCRIPT II.5 example program that calls SDBC functions and routines. This program simulates a bank with a single queue and multiple tellers and keeps track of simulation runs in a database. Each run is recorded as one row in a database table with the following definition:

```
CREATE TABLE BANKSIM
(RUNID   INTEGER NOT NULL PRIMARY KEY,
 TELLERS INTEGER NOT NULL,
 IATIME  REAL    NOT NULL,
 SRVTIME REAL    NOT NULL,
 UTIL    REAL,
 AVGQLEN REAL,
 MAXQLEN INTEGER)
```

RUNID is an integer ID that uniquely identifies the run. The input parameters are recorded in columns **TELLERS**, **IATIME**, and **SRVTIME**. **TELLERS** is the number of tellers working at the bank. The interarrival time of customers is exponentially distributed with a mean of **IATIME** minutes. The time required for a teller to serve a customer is exponentially distributed with a mean of **SRVTIME** minutes. The results of the run are stored in columns **UTIL**, **AVGQLEN**, and **MAXQLEN**. **UTIL** is the utilization of the tellers. **AVGQLEN** and **MAXQLEN** are the average and maximum length of the queue, respectively.

The **main** routine begins by prompting the user for the data source name, user name, and password, and then connects to the specified database. If the **BANKSIM** table does not exist, the **CREATE.TABLE** routine is called to create it. Then the **MAIN.LOOP** routine takes over and repeatedly displays a menu of choices, obtains the user's choice and processes it.

The user may choose to *Define a Run* by entering the ID and input parameters for a new run. A row is inserted into the **BANKSIM** table containing the specified **RUNID**, **TELLERS**, **IATIME**, and **SRVTIME**, with null values in the result columns, **UTIL**, **AVGQLEN**, and **MAXQLEN**.

The user may choose to *Execute a Run* by entering the ID of a defined run. The program obtains the input parameters for this run by retrieving its row. It then simulates one eight-hour day at the bank using these parameters. The results of the simulation are displayed to the user and saved in the **UTIL**, **AVGQLEN**, and **MAXQLEN** columns of the row.

The user may choose to *Show All Runs*; all rows of the **BANKSIM** table are retrieved and displayed, sorted by **RUNID**. The user may choose to *Delete a Run* by entering its ID; the row corresponding to this run is deleted from the **BANKSIM** table.

Finally, the user may choose to *Exit*, thereby terminating the **MAIN.LOOP** routine. The **main** routine then disconnects from the database and the program terminates.

```
preamble

''SDBC Example Program
''Single-Queue Multiple-Teller Bank Simulation
''Derived from Example 5 in the book,
''  "Building Simulation Models with SIMSCRIPT II.5"
''  by Edward C. Russell (CACI, 1983)

processes include GENERATOR and CUSTOMER

resources include TELLER

define MEAN.INTERARRIVAL.TIME, MEAN.SERVICE.TIME
as real variables

accumulate UTILIZATION as the average of N.X.TELLER
accumulate AVG.QUEUE.LENGTH as the average
       and MAX.QUEUE.LENGTH as the maximum of N.Q.TELLER

''SDBC Functions and Routines
define DB.AUTOCOMMIT.R as a          routine  given 1 argument
define DB.COMMIT.R     as a          routine  given 0 arguments
define DB.CONNECT.R    as a          routine  given 3 arguments
define DB.DISCONNECT.R as a          routine  given 0 arguments
define DB.EXISTS.F     as an integer function given 1 argument
define DB.FETCH.F      as an integer function given 0 arguments
define DB.GETINT.F     as an integer function given 1 argument
define DB.GETREAL.F    as a  double  function given 1 argument
define DB.GETTEXT.F    as a  text    function given 1 argument
define DB.NULL.F       as an integer function given 1 argument
define DB.QUERY.R      as a          routine  given 1 argument
define DB.ROLLBACK.R   as a          routine  given 0 arguments
define DB.SETINT.R     as a          routine  given 2 arguments
define DB.SETREAL.R    as a          routine  given 2 arguments
define DB.SETTEXT.R    as a          routine  given 2 arguments
define DB.UPDATE.F     as an integer function given 1 argument

end
```

```
main

define DSNAME, USERNAME, PASSWORD as text variables

write as "Enter data source name:", /
read DSNAME
write as "Enter user name:", /
read USERNAME
write as "Enter password:", /
read PASSWORD
call DB.CONNECT.R(DSNAME, USERNAME, PASSWORD)

if DB.EXISTS.F("BANKSIM") = 0 ''BANKSIM table does not exist
   call CREATE.TABLE          ''so create it
always

create every TELLER(1)

call MAIN.LOOP

call DB.DISCONNECT.R

end



routine CREATE.TABLE

define SQL as a text variable
define ROWS as an integer variable

''construct an SQL CREATE TABLE statement
SQL = CONCAT.F(
"CREATE TABLE BANKSIM ",
"(RUNID   INTEGER NOT NULL PRIMARY KEY,",
" TELLERS INTEGER NOT NULL,",
" IATIME  REAL    NOT NULL,",
" SRVTIME REAL    NOT NULL,",
" UTIL    REAL,",
" AVGQLEN REAL,",
" MAXQLEN INTEGER)")

''create the table
ROWS = DB.UPDATE.F(SQL)

end
```

```
routine MAIN.LOOP

define CHOICE as an integer variable

'DISPLAY.MENU'
print 7 lines thus

Enter
0 to Exit
1 to Define a Run
2 to Execute a Run
3 to Show All Runs
4 to Delete a Run

read CHOICE

select case CHOICE
   case 0  return
   case 1  call DEFINE.RUN
   case 2  call EXECUTE.RUN
   case 3  call SHOW.RUNS
   case 4  call DELETE.RUN
   default write as "Invalid choice", /
endselect

go to 'DISPLAY.MENU'

end
```

```
routine DEFINE.RUN

define RUNID, TELLERS, ROWS as integer variables
define IATIME, SRVTIME as real variables

write as /, "Enter Run #:", /
read RUNID

call DB.SETINT.R(1, RUNID)
call DB.QUERY.R("SELECT COUNT(*) FROM BANKSIM WHERE RUNID = ?")
if DB.FETCH.F = 1 and DB.GETINT.F(1) > 0
   write as "Run already defined", /
   return
otherwise

write as "Enter # of Tellers:", /
read TELLERS
write as "Enter Mean InterArrival Time in Minutes:", /
read IATIME
write as "Enter Mean Service Time in Minutes:", /
read SRVTIME

call DB.SETINT.R(1, RUNID)
call DB.SETINT.R(2, TELLERS)
call DB.SETREAL.R(3, IATIME)
call DB.SETREAL.R(4, SRVTIME)
ROWS = DB.UPDATE.F(CONCAT.F(
"INSERT INTO BANKSIM (RUNID, TELLERS, IATIME, SRVTIME)",
" VALUES (?, ?, ?, ?)"))

write as "Run defined", /

end
```

```
routine EXECUTE.RUN

define RUNID, MAXQLEN, ROWS as integer variables
define UTIL, AVGQLEN as real variables

write as /, "Enter Run #:", /
read RUNID

''lookup run definition
call DB.SETINT.R(1, RUNID)
call DB.QUERY.R(CONCAT.F(
"SELECT TELLERS, IATIME, SRVTIME, UTIL",
" FROM BANKSIM WHERE RUNID = ?"))

if DB.FETCH.F = 0 ''not found
   write as "Run undefined", /
   return
otherwise

if DB.NULL.F(4) = 1 ''UTIL is non-null
   write as "Run already executed", /
   return
otherwise

call SIMULATE.BANK given DB.GETINT.F(1), DB.GETREAL.F(2),
DB.GETREAL.F(3) yielding UTIL, AVGQLEN, MAXQLEN

''save run results
call DB.SETREAL.R(1, UTIL)
call DB.SETREAL.R(2, AVGQLEN)
call DB.SETINT.R(3, MAXQLEN)
call DB.SETINT.R(4, RUNID)
ROWS = DB.UPDATE.F(
"UPDATE BANKSIM SET UTIL=?,AVGQLEN=?,MAXQLEN=? WHERE RUNID=?")

end
```

```
routine SIMULATE.BANK given TELLERS, IATIME, SRVTIME
                      yielding UTIL, AVGQLEN, MAXQLEN

define TELLERS, MAXQLEN as integer variables
define IATIME, SRVTIME, UTIL, AVGQLEN as real variables

U.TELLER(1) = TELLERS
MEAN.INTERARRIVAL.TIME = IATIME
MEAN.SERVICE.TIME = SRVTIME

TIME.V = 0
reset totals of N.X.TELLER(1) and N.Q.TELLER(1)

activate a GENERATOR now

start simulation

UTIL = UTILIZATION(1) / TELLERS
AVGQLEN = AVG.QUEUE.LENGTH(1)
MAXQLEN = MAX.QUEUE.LENGTH(1)

write TELLERS as "# of Tellers:          ", I 3, /
write IATIME  as "Mean InterArrival Time: ", D(5,2),
" minutes", /
write SRVTIME as "Mean Service Time:      ", D(5,2),
" minutes", /
write UTIL    as "Teller Utilization:      ", D(4,2), /
write AVGQLEN as "Average Queue Length:  ", D(6,2), /
write MAXQLEN as "Maximum Queue Length:  ", I 3, /

end



process GENERATOR

''generate customer arrivals during one 8-hour day
while TIME.V < 8.0 / HOURS.V
do
   activate a CUSTOMER now
   wait EXPONENTIAL.F(MEAN.INTERARRIVAL.TIME, 1) minutes
loop

end



process CUSTOMER

request 1 TELLER
work EXPONENTIAL.F(MEAN.SERVICE.TIME, 2) minutes
relinquish 1 TELLER

end
```

```
routine SHOW.RUNS

''retrieve all rows sorted by ascending RUNID
call DB.QUERY.R("SELECT * FROM BANKSIM ORDER BY RUNID")

print 4 lines thus

                    Mean        Mean             Average  Maximum
                InterArrival  Service  Teller   Queue    Queue
Run#  #Tellers      Time        Time     Util.   Length   Length

while DB.FETCH.F = 1 ''for each row in BANKSIM
do
   if DB.NULL.F(5) = 1 ''UTIL is non-null
      print 1 line with DB.GETINT.F(1), DB.GETINT.F(2),
      DB.GETREAL.F(3), DB.GETREAL.F(4), DB.GETREAL.F(5),
      DB.GETREAL.F(6), DB.GETINT.F(7) thus
   *       *         *.*         *.*     *.**    *.**       *
   else ''this run has not been executed
      print 1 line with DB.GETINT.F(1), DB.GETINT.F(2),
      DB.GETREAL.F(3), DB.GETREAL.F(4) thus
   *       *         *.*         *.*
   always
loop

end



routine DELETE.RUN

define RUNID, ROWS as integer variables

write as /, "Enter Run #:", /
read RUNID

call DB.SETINT.R(1, RUNID)
ROWS = DB.UPDATE.F("DELETE FROM BANKSIM WHERE RUNID = ?")

if ROWS = 0 ''no rows were deleted
   write as "No such run", /
else
   write as "Run deleted", /
always

end
```

The following is a transcript from one execution of this program, starting with an empty database. User entries are italicized.

```
Enter data source name:
BANKSIMDB
Enter user name:
STEVE
Enter password:
SECRET

Enter
0 to Exit
1 to Define a Run
2 to Execute a Run
3 to Show All Runs
4 to Delete a Run
1

Enter Run #:
101
Enter # of Tellers:
2
Enter Mean InterArrival Time in Minutes:
5
Enter Mean Service Time in Minutes:
10
Run defined

Enter
0 to Exit
1 to Define a Run
2 to Execute a Run
3 to Show All Runs
4 to Delete a Run
3

                    Mean        Mean             Average  Maximum
                InterArrival  Service  Teller    Queue    Queue
Run#  #Tellers      Time        Time     Util.    Length   Length
 101       2         5.0        10.0
```

```
Enter
0 to Exit
1 to Define a Run
2 to Execute a Run
3 to Show All Runs
4 to Delete a Run
2

Enter Run #:
101
# of Tellers:             2
Mean InterArrival Time:  5.00 minutes
Mean Service Time:      10.00 minutes
Teller Utilization:       .96
Average Queue Length:    3.61
Maximum Queue Length:   13

Enter
0 to Exit
1 to Define a Run
2 to Execute a Run
3 to Show All Runs
4 to Delete a Run
3

                    Mean        Mean             Average  Maximum
                InterArrival  Service  Teller    Queue    Queue
Run#  #Tellers      Time        Time     Util.   Length   Length
 101       2         5.0        10.0      .96    3.61       13

Enter
0 to Exit
1 to Define a Run
2 to Execute a Run
3 to Show All Runs
4 to Delete a Run
1

Enter Run #:
102
Enter # of Tellers:
2
Enter Mean InterArrival Time in Minutes:
5
Enter Mean Service Time in Minutes:
10
Run defined
```

```
Enter
0 to Exit
1 to Define a Run
2 to Execute a Run
3 to Show All Runs
4 to Delete a Run
2

Enter Run #:
102
# of Tellers:            2
Mean InterArrival Time:  5.00 minutes
Mean Service Time:      10.00 minutes
Teller Utilization:       .90
Average Queue Length:    2.31
Maximum Queue Length:   10

Enter
0 to Exit
1 to Define a Run
2 to Execute a Run
3 to Show All Runs
4 to Delete a Run
3

                    Mean        Mean             Average  Maximum
                InterArrival  Service  Teller    Queue    Queue
Run#  #Tellers      Time        Time     Util.    Length   Length
 101       2         5.0        10.0      .96     3.61       13
 102       2         5.0        10.0      .90     2.31       10

Enter
0 to Exit
1 to Define a Run
2 to Execute a Run
3 to Show All Runs
4 to Delete a Run
4

Enter Run #:
101
Run deleted
```

```
Enter
0 to Exit
1 to Define a Run
2 to Execute a Run
3 to Show All Runs
4 to Delete a Run
3

                    Mean        Mean             Average  Maximum
                 InterArrival  Service  Teller  Queue    Queue
Run#  #Tellers      Time        Time     Util.  Length   Length
 102       2         5.0        10.0      .90    2.31      10

Enter
0 to Exit
1 to Define a Run
2 to Execute a Run
3 to Show All Runs
4 to Delete a Run
0
```

Chapter 7 Example Program: Job Shop Simulation

This section presents a complete SIMSCRIPT II.5 example program that calls SDBC functions and routines. This program simulates the operations of a job shop in which jobs arrive at random intervals and are processed by machines in the shop. The machines are grouped by type; for example, the shop may house eight drill presses, five lathes, and four polishing machines.

A job requires a sequence of tasks to be performed by machines in the shop. When the job arrives, it is sent to the machine group needed for the first task. If there is a unit currently available (idle) in this group, the task commences immediately using this unit; otherwise, the job waits in line for a unit to become available. Once the first task has finished, the job is sent to the machine group needed for the second task, and so on, until all of the tasks have been completed.

Each type of machine is described by one row in a database table named **Machines** with the following definition:

```
CREATE TABLE Machines
(Machine_ID      CHAR(2)     NOT NULL PRIMARY KEY,
 Machine_Name    VARCHAR(20) NOT NULL,
 Number_of_Units SMALLINT    NOT NULL)
```

Machine_ID is a two-character code that uniquely identifies the machine type. **Machine_Name** gives the name of the machine type, and **Number_of_Units** specifies the number of units of this type in the shop.

The program assumes that the **Machines** table has already been created and populated with rows. For example, the contents of the table might look like this:

Machine_ID	Machine_Name	Number_of_Units
CU	Casting Units	14
DP	Drill Presses	8
LA	Lathes	5
PL	Planes	4
PM	Polishing Machines	4
SH	Shapers	16

The shop will only process jobs of a certain type. The accepted job types are described in a database table named **Job_Types** with the following definition:

```
CREATE TABLE Job_Types
(Job_Type_Number   SMALLINT NOT NULL,
 Sequence_Number   SMALLINT NOT NULL,
 Machine_ID        CHAR(2)  NOT NULL REFERENCES Machines,
 Mean_Service_Time REAL     NOT NULL,
 PRIMARY KEY (Job_Type_Number, Sequence_Number))
```

Each row of this table describes one task of the job type identified by **Job_Type_Number**. The task requires the use of one unit of machine type **Machine_ID** for a random number of hours that is exponentially distributed with a mean of **Mean_Service_Time**. **Sequence_Number** is used to specify the order of tasks for a given job type. The combination of **Job_Type_Number** and **Sequence_Number** uniquely identifies a row and is designated as the primary key. **Machine_ID** is declared as a *foreign key* by the **REFERENCES** clause, which guarantees that its value is present in the **Machine_ID** column of the **Machines** table.

The program assumes that the **Job_Types** table has already been created and populated with rows. For example, the following table describes the tasks of three job types: Job Type 117 (four tasks), Job Type 123 (three tasks), and Job Type 125 (five tasks).

Job_Type_Number	Sequence_Number	Machine_ID	Mean_Service_Time
117	1	CU	2.0833
117	2	PL	0.5833
117	3	LA	0.3333
117	4	PM	1.0000
123	1	SH	1.7500
123	2	DP	1.5000
123	3	LA	1.0833
125	1	CU	3.9166
125	2	SH	4.1666
125	3	DP	0.8333
125	4	PL	0.5000
125	5	PM	0.4166

One simulation run measures the utilization of each machine group and the average and maximum number of jobs waiting for each group. This data is stored in a database table named **Results** with the following definition. (This table is assumed by the program to exist.)

```
CREATE TABLE Results
(Run_Number  SMALLINT NOT NULL,
 Machine_ID  CHAR(2)  NOT NULL REFERENCES Machines,
 Utilization REAL     NOT NULL,
 Avg_Backlog REAL     NOT NULL,
 Max_Backlog INTEGER  NOT NULL,
 PRIMARY KEY (Run_Number, Machine_ID))
```

The **main** routine begins by prompting the user for the data source name, user name, and password, and then connects to the specified database. The user then enters a run number. If results for this run can be found in the **Results** table, they are retrieved and displayed to the user and no simulation is performed; otherwise, the program prepares to run a new simulation.

First, the **SETUP.MACHINES** routine reads the machine types from the **Machines** table. Second, the **SETUP.JOB.TYPES** routine reads the job types from the **Job_Types** table and stores them as a set of job types where each job type owns a set of its tasks. In addition, this routine prompts the user to enter the probability of each job type. Third, the program prompts the user to enter the mean job interarrival time and duration of the simulation, and then begins the simulation.

When the simulation has finished, the **SAVE.RESULTS** routine inserts the results atomically into the **Results** table, so that either all or none of the results are saved. The **SHOW.RESULTS** routine retrieves the results from the database and displays them to the user. Lastly, the program disconnects from the database before terminating.

```
preamble

''SDBC Example Program
''Job Shop Simulation
''Derived from Example 6 in the book,
''  "Building Simulation Models with SIMSCRIPT II.5"
''  by Edward C. Russell (CACI, 1983)

processes include GENERATOR and JOB

resources

every MACHINE
   has a MACHINE.ID,
        a MACHINE.NAME,
   and a NUMBER.OF.UNITS
define MACHINE.ID, MACHINE.NAME as text variables
define NUMBER.OF.UNITS as an integer variable

temporary entities

every TASK
   has a MACHINE.INDEX
   and a MEAN.SERVICE.TIME
   and belongs to a TASK.SEQUENCE
define MACHINE.INDEX as an integer variable
define MEAN.SERVICE.TIME as a real variable

every JOB.TYPE
   has a JOB.TYPE.NUMBER,
   owns a TASK.SEQUENCE,
   and belongs to the JOB.TYPE.LIST
define JOB.TYPE.NUMBER as an integer variable

the system
   has a RUN.NUMBER,
        a MEAN.INTERARRIVAL.TIME,
        a STOP.TIME,
   and a JOB.MIX random step variable
   and owns the JOB.TYPE.LIST
define RUN.NUMBER as an integer variable
define MEAN.INTERARRIVAL.TIME, STOP.TIME as real variables
define JOB.MIX as an integer, stream 9 variable

define TASK.SEQUENCE, JOB.TYPE.LIST as FIFO sets

accumulate UTILIZATION as the average of N.X.MACHINE
accumulate AVG.BACKLOG as the average
       and MAX.BACKLOG as the maximum of N.Q.MACHINE

define HOURS to mean units
```

```
''SDBC Functions and Routines
define DB.AUTOCOMMIT.R as a          routine  given 1 argument
define DB.COMMIT.R     as a          routine  given 0 arguments
define DB.CONNECT.R    as a          routine  given 3 arguments
define DB.DISCONNECT.R as a          routine  given 0 arguments
define DB.EXISTS.F     as an integer function given 1 argument
define DB.FETCH.F      as an integer function given 0 arguments
define DB.GETINT.F     as an integer function given 1 argument
define DB.GETREAL.F    as a  double  function given 1 argument
define DB.GETTEXT.F    as a  text    function given 1 argument
define DB.NULL.F       as an integer function given 1 argument
define DB.QUERY.R      as a          routine  given 1 argument
define DB.ROLLBACK.R   as a          routine  given 0 arguments
define DB.SETINT.R     as a          routine  given 2 arguments
define DB.SETREAL.R    as a          routine  given 2 arguments
define DB.SETTEXT.R    as a          routine  given 2 arguments
define DB.UPDATE.F     as an integer function given 1 argument

end
```

```
main

define DSNAME, USERNAME, PASSWORD as text variables

write as "Enter data source name:", /
read DSNAME
write as "Enter user name:", /
read USERNAME
write as "Enter password:", /
read PASSWORD
call DB.CONNECT.R(DSNAME, USERNAME, PASSWORD)

write as /, "Enter Run #:", /
read RUN.NUMBER

call DB.SETINT.R(1, RUN.NUMBER)
call DB.QUERY.R(
"SELECT COUNT(*) FROM Results WHERE Run_Number = ?")
if DB.FETCH.F = 1 and DB.GETINT.F(1) > 0 ''this is an old run
   go to 'FINISH' ''display results of old run
otherwise

''simulate new run
call SETUP.MACHINES
call SETUP.JOB.TYPES

write as /, "Enter mean job interarrival time in hours:", /
read MEAN.INTERARRIVAL.TIME
write as "Enter duration of simulation in hours:", /
read STOP.TIME

activate a GENERATOR now
start simulation

'FINISH'
call SHOW.RESULTS
call DB.DISCONNECT.R

write as /, "Press return to exit", /
read as /

end
```

```
routine SETUP.MACHINES

''retrieve machine information from the database and
''use it to initialize the MACHINE resource

''first determine the number of machine groups
call DB.QUERY.R("SELECT COUNT(*) FROM Machines")
if DB.FETCH.F = 1 ''should always be true
   create every MACHINE(DB.GETINT.F(1))
always

''then obtain the information for each machine group
write as /, "Machines:", /
call DB.QUERY.R("SELECT * FROM Machines ORDER BY Machine_Name")
for each MACHINE while DB.FETCH.F = 1
do
   MACHINE.ID(MACHINE) = DB.GETTEXT.F(1)
   MACHINE.NAME(MACHINE) = DB.GETTEXT.F(2)
   NUMBER.OF.UNITS(MACHINE) = DB.GETINT.F(3)
   U.MACHINE(MACHINE) = NUMBER.OF.UNITS(MACHINE)
   write NUMBER.OF.UNITS(MACHINE), MACHINE.NAME(MACHINE)
   as I 3, " ", T *, /
loop

end
```

```
routine SETUP.JOB.TYPES

define JOB.TYPE, TASK as pointer variables
define PROBABILITY as a real variable

''retrieve job types and their tasks in sequence
call DB.QUERY.R("SELECT * FROM Job_Types ORDER BY 1, 2")
while DB.FETCH.F = 1 ''for each row in Job_Types
do
   if JOB.TYPE.LIST is empty or
   DB.GETINT.F(1) > JOB.TYPE.NUMBER(JOB.TYPE)
      ''encountered a new job type
      create a JOB.TYPE
      JOB.TYPE.NUMBER(JOB.TYPE) = DB.GETINT.F(1)
      file JOB.TYPE in JOB.TYPE.LIST
      write JOB.TYPE.NUMBER(JOB.TYPE)
      as /, "Job Type ", I 3, ":", /
   always
   ''save task information
   create a TASK
   for each MACHINE with MACHINE.ID(MACHINE) = DB.GETTEXT.F(3)
      find the first case
   MACHINE.INDEX(TASK) = MACHINE
   MEAN.SERVICE.TIME(TASK) = DB.GETREAL.F(4)
   file TASK in TASK.SEQUENCE(JOB.TYPE)
   write MEAN.SERVICE.TIME(TASK), MACHINE.NAME(MACHINE)
   as D(7,4), " hours on ", T *, /
loop

''prompt the user to enter job type probabilities and
''use them to initialize the JOB.MIX random step variable
write as /
for each JOB.TYPE in JOB.TYPE.LIST
do
   write JOB.TYPE.NUMBER(JOB.TYPE)
   as "Enter probability of Job Type ", I 3, ":", /
   read PROBABILITY
   write PROBABILITY, JOB.TYPE.NUMBER(JOB.TYPE)
   as D(5,3), " ", I 3, " " using the buffer
loop
write as "*" using the buffer ''marks the end of the input
read JOB.MIX using the buffer ''initialize random step variable

end
```

```
process GENERATOR

while TIME.V < STOP.TIME
do
   activate a JOB now
   wait EXPONENTIAL.F(MEAN.INTERARRIVAL.TIME, 10) HOURS
loop

call SAVE.RESULTS

end



process JOB

define TYPE.NUMBER as an integer variable
define JOB.TYPE, TASK as pointer variables

TYPE.NUMBER = JOB.MIX ''randomly generate the job type
for each JOB.TYPE in JOB.TYPE.LIST
with JOB.TYPE.NUMBER(JOB.TYPE) = TYPE.NUMBER
   find the first case

''perform the tasks for this job type in sequence
for each TASK in TASK.SEQUENCE(JOB.TYPE)
do
   request 1 unit of MACHINE(MACHINE.INDEX(TASK))
   work EXPONENTIAL.F(MEAN.SERVICE.TIME(TASK),
   MIN.F(MACHINE.INDEX(TASK), 10)) HOURS
   relinquish 1 unit of MACHINE(MACHINE.INDEX(TASK))
loop

end
```

```
routine SAVE.RESULTS

call DB.AUTOCOMMIT.R(0) ''turn Auto-Commit OFF

''atomically insert the result rows, one for each machine group
for each MACHINE
do
   call DB.SETINT.R(1, RUN.NUMBER)
   call DB.SETTEXT.R(2, MACHINE.ID(MACHINE))
   call DB.SETREAL.R(3,
   UTILIZATION(MACHINE) / NUMBER.OF.UNITS(MACHINE))
   call DB.SETREAL.R(4, AVG.BACKLOG(MACHINE))
   ''although the mode of MAX.BACKLOG is double, it will be
   ''converted to integer when stored in column Max_Backlog
   call DB.SETREAL.R(5, MAX.BACKLOG(MACHINE))
   if DB.UPDATE.F("INSERT INTO Results VALUES (?,?,?,?,?)")<>1
      call DB.ROLLBACK.R ''error - undo all insertions
      go to 'EXIT'
   otherwise
loop

call DB.COMMIT.R ''success - make all insertions permanent

'EXIT'
call DB.AUTOCOMMIT.R(1) ''turn Auto-Commit ON

end
```

```
routine SHOW.RESULTS

define JOIN as a text variable

''construct SQL statement to join Machines and Results tables
JOIN = CONCAT.F(
"SELECT Machine_Name, Number_of_Units, Utilization,",
"       Avg_Backlog, Max_Backlog",
" FROM  Machines, Results",
" WHERE Machines.Machine_ID = Results.Machine_ID",
"   AND Run_Number = ?",
" ORDER BY Machine_Name")

''execute the query
call DB.SETINT.R(1, RUN.NUMBER)
call DB.QUERY.R(JOIN)

''fetch and display the results of the run
print 4 lines with RUN.NUMBER thus

Results of Run #  *:
                                          Average    Maximum
Machine                 #Units    Util.   Backlog    Backlog

while DB.FETCH.F = 1
   print 1 line with DB.GETTEXT.F(1), DB.GETINT.F(2),
   DB.GETREAL.F(3), DB.GETREAL.F(4), DB.GETINT.F(5) thus
********************       *      *.**      *.**         *

end
```

Assume that the **Machines**, **Job_Types**, and **Results** tables have already been created and that the **Machines** and **Job_Types** tables have been populated with the contents shown at the beginning of this section. The following is a transcript from one execution of this program. User entries are italicized.

```
Enter data source name:
JOBSHOPSIMDB
Enter user name:
STEVE
Enter password:
SECRET

Enter Run #:
1

Machines:
 14 Casting Units
  8 Drill Presses
  5 Lathes
  4 Planes
  4 Polishing Machines
 16 Shapers

Job Type 117:
 2.0833 hours on Casting Units
  .5833 hours on Planes
  .3333 hours on Lathes
 1.0000 hours on Polishing Machines

Job Type 123:
 1.7500 hours on Shapers
 1.5000 hours on Drill Presses
 1.0833 hours on Lathes

Job Type 125:
 3.9166 hours on Casting Units
 4.1666 hours on Shapers
  .8333 hours on Drill Presses
  .5000 hours on Planes
  .4166 hours on Polishing Machines

Enter probability of Job Type 117:
.241
Enter probability of Job Type 123:
.44
Enter probability of Job Type 125:
.319

Enter mean job interarrival time in hours:
.16
Enter duration of simulation in hours:
```

```
40

Results of Run #  1:

                                          Average    Maximum
Machine                 #Units    Util.   Backlog    Backlog
Casting Units             14       .57      .01         2
Drill Presses              8       .62      .25         7
Lathes                     5       .65      .63        10
Planes                     4       .37      .02         2
Polishing Machines         4       .48      .17         3
Shapers                   16       .66      .12         6

Press return to exit
```

APPENDIX A SDBC Functions and Routines

Routine **DB.AUTOCOMMIT.R(SETTING)**

SETTING: 0 or 1, mode is **INTEGER**

Turns Auto-Commit OFF if **SETTING** is 0; otherwise, turns Auto-Commit ON.

Routine **DB.COMMIT.R**

Terminates and commits the current transaction.

Routine **DB.CONNECT.R(DSNAME, USERNAME, PASSWORD)**

DSNAME: data source name, mode is **TEXT**
USERNAME: database user name, mode is **TEXT**
PASSWORD: database password, mode is **TEXT**

Connects to the database identified by the named ODBC data source using the given user name and password.

Routine **DB.DISCONNECT.R**

Disconnects from the database.

Function **DB.EXISTS.F(TABLE)**

TABLE: database table name, mode is **TEXT**
return value: 0 or 1, mode is **INTEGER**

Returns 1 if the named table exists, or returns 0 if the table does not exist.

Function **DB.FETCH.F**

return value: 0 or 1, mode is **INTEGER**

Retrieves the next row of the query result and returns 1, or returns 0 if there are no more rows.

Function **DB.GETINT.F(COLUMN)**

COLUMN: column number, mode is **INTEGER**
return value: column value, mode is **INTEGER**

Returns the **INTEGER** value of the specified column in the current row.

Function **DB.GETREAL.F(COLUMN)**

COLUMN: column number, mode is **INTEGER**
return value: column value, mode is **DOUBLE**

Returns the **DOUBLE** value of the specified column in the current row.

Function **DB.GETTEXT.F(COLUMN)**

COLUMN: column number, mode is **INTEGER**
return value: column value, mode is **TEXT**

Returns the **TEXT** value of the specified column in the current row.

Function **DB.NULL.F(COLUMN)**

COLUMN: column number, mode is **INTEGER**
return value: 0 or 1, mode is **INTEGER**

Returns 0 if the value of the specified column in the current row is null, or returns 1 if the value is non-null.

Routine **DB.QUERY.R(COMMAND)**

COMMAND: SQL query statement, mode is **TEXT**

Executes the given SQL query statement.

Routine **DB.ROLLBACK.R**

Terminates and rolls back the current transaction.

Routine **DB.SETINT.R(PARM, VALUE)**

PARM: parameter number, mode is **INTEGER**
VALUE: parameter value, mode is **INTEGER**

Sets the specified parameter to the given **INTEGER** value.

Routine **DB.SETREAL.R(PARM, VALUE)**

PARM: parameter number, mode is **INTEGER**
VALUE: parameter value, mode is **DOUBLE**

Sets the specified parameter to the given **DOUBLE** value.

Routine **DB.SETTEXT.R(PARM, VALUE)**

PARM: parameter number, mode is **INTEGER**
VALUE: parameter value, mode is **TEXT**

Sets the specified parameter to the given **TEXT** value.

Function **DB.UPDATE.F(COMMAND)**

COMMAND: SQL update statement, mode is **TEXT**
return value: number of affected rows, mode is **INTEGER**

Executes the given SQL update statement, and returns the number of affected rows if applicable.

APPENDIX B SQL Syntax

SDBC supports, at a minimum, the following SQL syntax based on the Entry Level of the ANSI SQL-92 standard. Additional SQL features provided by the DBMS can also be used; see the DBMS documentation for information.

Notation: SQL key words and special characters are in **BOLD**
Syntactic placeholders are in *ITALICS*
Mandatory elements are in { braces }
Optional elements are in [brackets]
Alternatives are separated by |
Lists of one or more elements, separated by commas, are denoted by *

Argument to **DB.UPDATE.F**:

CREATE_TABLE | DROP_TABLE | INSERT | UPDATE | DELETE

Argument to **DB.QUERY.R**:

TABLE_EXPR
[**ORDER BY** { { *COLUMN* | *NUMBER* } [**ASC** | **DESC**] }*]

CREATE_TABLE: **CREATE TABLE** *TABLE* **(** *TDEF** **)**

TDEF: *COLDEF* |
{ **PRIMARY KEY** | **UNIQUE** } **(** *COLUMN** **)** |
FOREIGN KEY (*COLUMN** **)**
REFERENCES *TABLE* [**(** *COLUMN** **)**]

COLDEF: *COLUMN DATATYPE* [**NOT NULL**]
[**PRIMARY KEY** | **UNIQUE**]
[**REFERENCES** *TABLE* [**(** *COLUMN* **)**]]

DATATYPE: **SMALLINT** | **INTEGER** |
REAL | **DOUBLE** [**PRECISION**] |
CHAR(*NUMBER* **)** | **VARCHAR(** *NUMBER* **)**

DROP_TABLE: **DROP TABLE** *TABLE* [**RESTRICT** | **CASCADE**]

INSERT: **INSERT INTO** *TABLE* [**(** *COLUMN** **)**]
{ **VALUES (** { *EXPR* | **NULL** }* **)** | *TABLE_EXPR* }

UPDATE: **UPDATE** *TABLE*
SET { *COLUMN* **=** { *EXPR* | **NULL** } }*
[**WHERE** *CONDITION*]

DELETE: **DELETE FROM** *TABLE*
[**WHERE** *CONDITION*]

TABLE_EXPR: [*TABLE_EXPR* **UNION** [**ALL**]]
{ *SELECT* | **(** *TABLE_EXPR* **)** }

SELECT: **SELECT** [**ALL** | **DISTINCT**] { ***** | { *EXPR* [**AS** *COLUMN*] }*
}
FROM { *TABLE* [*RANGE_VAR*] }*
[**WHERE** *CONDITION*]
[**GROUP BY** *COLREF**]
[**HAVING** *CONDITION*]

CONDITION: *CTERM* | *CONDITION* **OR** *CTERM*

CTERM: CFACTOR | CTERM **AND** CFACTOR

CFACTOR: [**NOT**] { *COMPARE* | *IN* | *BETWEEN* | *EXISTS* | *NULL* | *LIKE* | **(** *CONDITION* **)** }

COMPARE: *EXPR* { **<** | **<=** | **=** | **<>** | **>** | **>=** }
{ *EXPR* | { **ALL** | **ANY** | **SOME** } **(** *TABLE_EXPR* **)** }

IN: *EXPR* [**NOT**] **IN (** *TABLE_EXPR* | *EXPR** **)**

BETWEEN: *EXPR* [**NOT**] **BETWEEN** *EXPR* **AND** *EXPR*

EXISTS: **EXISTS (** TABLE_EXPR **)**

NULL: *COLREF* **IS** [**NOT**] **NULL**

LIKE: *COLREF* [**NOT**] **LIKE** *PATTERN* [**ESCAPE** *STRING*]

PATTERN: a character string pattern enclosed in single quotes in which each underscore matches any single character and each

percent sign (`%`)
matches any sequence of zero or more characters

EXPR: *TERM* | *EXPR* { **+** | **-** } *TERM*

TERM: *FACTOR* | *TERM* { ***** | **/** } *FACTOR*

FACTOR: [**+** | **-**] { *FUNCTION* | *COLREF* | *NUMBER* | *STRING* | **(** *TABLE_EXPR* **)** | **(** *EXPR* **)** }

FUNCTION: **COUNT(** ***** | **DISTINCT** *COLREF* **)** | { **AVG** | **MAX** | **MIN** | **SUM** } **(** [**ALL**] *EXPR* | **DISTINCT** *COLREF* **)**

COLREF: [{ TABLE | RANGE_VAR } .] COLUMN

TABLE: NAME

RANGE_VAR: NAME

COLUMN: NAME

NAME: a case-insensitive identifier composed of a letter followed by zero or more letters, digits, and underscores; examples:

```
address     P          S52a
Last_Name   EMP_ID     COL2
```

NUMBER: an integer or real constant with optional sign, and with optional scientific notation; examples:

```
5           0.7        -1058
+70.1389    2E12       -.43E-6
```

STRING: a character string enclosed in single quotes; examples:

```
'Hey!'      'a'        'NEW MEXICO'
'don''t'    ''         '16 lbs.'
```

APPENDIX C SQLSTATE Codes

The first value appearing in brackets within an SDBC run-time error message is a five-character SQLSTATE code. Most of these codes are defined by X/*Open Data Management: Structured Query Language (SQL), Version 2* (March 1995); however, additional codes may be defined by the ODBC driver. The following is a partial list of SQLSTATE codes and their meanings.

01000 General warning
01001 Cursor operation conflict
01002 Disconnect error
01003 NULL value eliminated in set function
01004 String data, right truncated
01006 Privilege not revoked
01007 Privilege not granted
01S00 Invalid connection string attribute
01S01 Error in row
01S02 Option value changed
01S06 Attempt to fetch before the result set returned the first rowset
01S07 Fractional truncation
01S08 Error saving File DSN
01S09 Invalid keyword

07002 COUNT field incorrect
07005 Prepared statement not a cursor-specification
07006 Restricted data type attribute violation
07009 Invalid descriptor index
07S01 Invalid use of default parameter

08001 Client unable to establish connection
08002 Connection name in use
08003 Connection does not exist
08004 Server rejected the connection
08007 Connection failure during transaction
08S01 Communication link failure

21S01 Insert value list does not match column list
21S02 Degree of derived table does not match column list

22001 String data, right truncated
22002 Indicator variable required but not supplied
22003 Numeric value out of range

22007 Invalid datetime format
22008 Datetime field overflow
22012 Division by zero
22015 Interval field overflow
22018 Invalid character value for cast specification
22019 Invalid escape character
22025 Invalid escape sequence
22026 String data, length mismatch

23000 Integrity constraint violation

24000 Invalid cursor state

25000 Invalid transaction state
25S01 Transaction state
25S02 Transaction is still active
25S03 Transaction is rolled back

28000 Invalid authorization specification

34000 Invalid cursor name

3C000 Duplicate cursor name

3D000 Invalid catalog name

3F000 Invalid schema name

40001 Serialization failure
40003 Statement completion unknown

42000 Syntax error or access violation
42S01 Base table or view already exists
42S02 Base table or view not found
42S11 Index already exists
42S12 Index not found
42S21 Column already exists
42S22 Column not found

44000 WITH CHECK OPTION violation

HY000 General error
HY001 Memory allocation error
HY003 Invalid application buffer type
HY004 Invalid SQL data type
HY007 Associated statement is not prepared

HY008 Operation canceled
HY009 Invalid use of null pointer
HY010 Function sequence error
HY011 Attribute cannot be set now
HY012 Invalid transaction operation code
HY013 Memory management error
HY014 Limit on the number of handles exceeded
HY015 No cursor name available
HY016 Cannot modify an implementation row descriptor
HY017 Invalid use of an automatically allocated descriptor handle
HY018 Server declined cancel request
HY019 Non-character and non-binary data sent in pieces
HY020 Attempt to concatenate a null value
HY021 Inconsistent descriptor information
HY024 Invalid attribute value
HY090 Invalid string or buffer length
HY091 Invalid descriptor field identifier
HY092 Invalid attribute/option identifier
HY093 Invalid parameter number
HY095 Function type out of range
HY096 Invalid information type
HY097 Column type out of range
HY098 Scope type out of range
HY099 Nullable type out of range
HY100 Uniqueness option type out of range
HY101 Accuracy option type out of range
HY103 Invalid retrieval code
HY104 Invalid precision or scale value
HY105 Invalid parameter type
HY106 Fetch type out of range
HY107 Row value out of range
HY109 Invalid cursor position
HY110 Invalid driver completion
HY111 Invalid bookmark value
HYC00 Optional feature not implemented
HYT00 Timeout expired
HYT01 Connection timeout expired

IM001 Driver does not support this function
IM002 Data source name not found and no default driver specified
IM003 Specified driver could not be loaded
IM004 Driver's SQLAllocHandle on SQL_HANDLE_ENV failed
IM005 Driver's SQLAllocHandle on SQL_HANDLE_DBC failed
IM006 Driver's SQLSetConnectAttr failed
IM007 No data source or driver specified; dialog prohibited
IM008 Dialog failed

IM009 Unable to load translation DLL
IM010 Data source name too long
IM011 Driver name too long
IM012 DRIVER keyword syntax error
IM013 Trace file error
IM014 Invalid name of File DSN
IM015 Corrupt file data source

INDEX