38

Chapter 38

SQL Sessions

An SQL-session spans the execution of one or more consecutive SQL statements by a single user. In order to execute any SQL statements, your DBMS has to establish an SQL-Connection (using the CONNECT statement) — the SQL-session thus initiated is then associated with that SQL-Connection. If you don't explicitly do a CONNECT statement on your own, your DBMS will effectively execute a default CONNECT statement to establish an SQL-Connection for you — this is known as the *default SQL-Connection*.

An SQL-Connection has two possible states: it is either current or dormant. Only one SQL-Connection (and its associated SQL-session) can be current at a time. An SQL-Connection is initially established by a CONNECT statement as the current SQL-Connection and it remains the current SQL-Connection unless and until another CONNECT statement or a SET CONNECTION statement puts it (and its associated SQL-session) into a dormant state by establishing another SQL-Connection as the current SQL-Connection. When would you use SET CONNECTION? Well, the SQL Standard says that your DBMS must support at least one SQL-Connection — but it may support more than one concurrent SQL-Connection. In the latter case, your application program may connect to more than one SQL-server, selecting the one it wants to use (the current or active SQL-Connection) with a SET CONNECTION statement.

An SQL-Connection ends either when you issue a DISCONNECT statement or in some implementation-defined way following the last call to an <externally-invoked procedure> within the last active SQL-client Module.

Every SQL-session has a user <AuthorizationID> to provide your DBMS with an <AuthorizationID> for Privilege checking during operations on SQL-data. You can specify this

<AuthorizationID> with the CONNECT statement or allow it to default to an <AuthorizationID> provided by your DBMS. You can also change the <AuthorizationID> during the SQL-session.

Every SQL-session also has a default local time zone offset to provide your DBMS with a time zone offset when a time or a timestamp value needs one. When you begin an SQL-session, your DBMS sets the default time zone offset to a value chosen by your vendor. You can change this to a more appropriate value with the SET TIME ZONE statement.

Every SQL-session has what the SQL Standard calls "enduring characteristics" — these all have initial default values at the beginning of an SQL-session, but you change any of them with the SET SESSION CHARACTERISTICS statement. SQL-sessions also have a "context" — characteristics of the SQL-session that your DBMS preserves when an SQL-session is made dormant so that it can restore the SQL-session properly when it is made current again. The context of an SQL-session includes the current SESSION_USER, the CURRENT_USER, the CURRENT_ROLE, the CURRENT_PATH, the identities of all temporary Tables, the current default time zone offset, the current constraint mode for all Constraints, the current transaction access mode, the position of all open Cursors, the current transaction isolation level, the current transaction diagnostics area limit, the value of all valid locators, and information of any active SQL-invoked routines.

SQL-Connections

You can establish an SQL-Connection either explicitly, by issuing a CONNECT statement; or implicitly, by invoking a procedure that works on SQL-data when there is no current SQL-session (in this case, your DBMS acts as if you explicitly issued CONNECT TO DEFAULT;). You can change the state of an SQL-Connection (from current to dormant and back again) with the SET CONNECTION statement. You can also end an SQL-Connection (and therefore its associated SQL-session) with the DISCONNECT statement. Here's how.

CONNECT Statement

The CONNECT statement explicitly establishes an SQL-Connection. The establishment of an SQL-Connection gives you access to the SQL-data on an SQL-server in your environment and thus starts an SQL-session. The required syntax for the CONNECT statement is as follows.

```
CONNECT TO
   DEFAULT |
   <SQL-server name> [ AS <Connection name> ] [ USER <AuthorizationID> ]
```

The CONNECT statement may not be executed during a transaction unless your DBMS supports transactions that affect multiple SQL-servers.

DEFAULT Connection

The SQL Standard does not require an explicit CONNECT statement to start an SQL-session. If the first SQL statement in an SQL-session is anything other than a CONNECT statement, your DBMS will first execute this SQL statement:

```
CONNECT TO DEFAULT;
```

to establish the default SQL-Connection before proceeding further. You can also issue an explicit CONNECT TO DEFAULT; statement if you want to deliberately establish your DBMS's default SQL-Connection. In either case, if the default SQL-Connection has already been established, (e.g., it was already the subject of a CONNECT statement or a SET CONNECTION statement and no DISCONNECT statement has been issued for it) CONNECT will fail; your DBMS will return the SQLSTATE error 08002 "connection exception-connection name in use."

[NON-PORTABLE] The effect of CONNECT TO DEFAULT; is non-standard because the SQL Standard requires implementors to define the default SQL-Connection and the default SQL-server. [OCELOT Implementation] The OCELOT DBMS that comes with this book treats the (explicit or implicit) SQL statement:

```
CONNECT TO DEFAULT;
```

as equivalent to this SQL statement:

```
CONNECT TO 'ocelot' AS 'ocelot' USER 'ocelot';
```

This CONNECT statement causes the OCELOT DBMS to establish a SQL-Connection to a default Cluster (or default SQL-server) called OCELOT, using a default <Connection name> of OCELOT and a default user <AuthorizationID> of OCELOT. If the default Cluster can't be found, the DBMS will create it.

<SQL-Server name> Clause

The other form of the CONNECT statement has three possible arguments, only one of which is mandatory. The syntax CONNECT TO <SQL-server name>; e.g.,

```
CONNECT TO 'some_server';
```

establishes an SQL-Connection to the SQL-server named. (Remember that the SQL-server is that portion of your environment that actually carries out the database operations.) <SQL-server name> is either a <character string literal>, a <host parameter name>, or an <SQL parameter reference> whose value represents a valid <SQL-server name>. The SQL Standard is deliberately vague about just what an SQL-server is and consequently leaves the method for determining its location and the communication protocol required to access it up to the DBMS.

AS Clause

The syntax CONNECT TO <SQL-server name> AS <Connection name>; e.g.,

```
CONNECT TO 'some_server' AS 'connection_1';
```

establishes an SQL-Connection named connection_1 to the SQL-server named. <Connection name> is a <simple value specification>, (e.g., a <character string literal>, <host parameter name>, or <SQL parameter reference>) whose value represents a valid <Connection name>. (If <Connection name> does not evaluate to a valid <Connection name>, CONNECT will fail; your DBMS will return the SQLSTATE error 2E000 "invalid connection name.")

[NON-PORTABLE] A <Connection name> must be a <regular identifier> or a <delimited identifier> that is no more than 128 octets in length, but the value of a valid <Connection name> is non-standard because the SQL Standard requires implementors to define what a valid <Connection name> may be and to which Character set <Connection name>s belong.

[OCELOT Implementation] The OCELOT DBMS that comes with this book defines a <Connection name> as any valid <regular identifier> or <delimited identifier> whose characters belong to the INFORMATION_SCHEMA.SQL_TEXT Character set.

If your CONNECT statement doesn't include the optional AS <Connection name> clause, the value of <Connection name> defaults to the value of <SQL-server name>. The following SQL statements are therefore equivalent (assuming that the default SQL-Connection is to an SQL-server named some_server):

```
CONNECT TO DEFAULT;
CONNECT TO 'some_server';
CONNECT TO 'some_server' AS 'some_server';
```

NOTE: The AS clause can only be omitted from the first CONNECT statement issued for a particular SQL-server. On the second and subsequent Connections, an explicit <Connection name> must be provided to your DBMS because <Connection name>s must be unique for the entire SQL-environment at any given time. You'll use the <Connection name> with the SET CONNECTION statement to switch between different SQL-Connections. If <Connection name> evaluates to a <Connection name> that is already in use — e.g., it was the subject of CONNECT TO or SET CONNECTION and DISCONNECT has not been issued for it — CONNECT TO will fail; the DBMS will return the SQLSTATE error 08002 "connection exception-connection name in use."

USER Clause

The syntax CONNECT TO <SQL-server name> USER <AuthorizationID>; e.g.,

```
CONNECT TO 'some_server' USER 'bob';
```

establishes an SQL-Connection, with an SQL-session <AuthorizationID> of bob, to the SQL-server named. <AuthorizationID> is a <simple value specification> — e.g., a <character string literal>, <host parameter name>, or <SQL parameter reference> — whose value represents a valid <AuthorizationID>. (If <AuthorizationID> does not evaluate to a valid user <AuthorizationID>, CONNECT will fail; your DBMS will return the SQLSTATE error 28000 "invalid authorization specification.")

If your CONNECT statement doesn't include the optional USER <AuthorizationID> clause, the value of the SQL-session user defaults to an <AuthorizationID> chosen by your DBMS. The following SQL statements are therefore equivalent (assuming that the default SQL-Connection is to an SQL-server named some_server):

```
CONNECT TO DEFAULT;
CONNECT TO 'some_server';
CONNECT TO 'some_server' AS 'some_server' USER 'default_user';
```

[NON-PORTABLE] The effect of omitting the optional USER clause from a CONNECT statement is non-standard because the SQL Standard requires implementors to define their own initial default SQL-session <AuthorizationID>. [OCELOT Implementation] The OCELOT DBMS that comes with this book has an initial default <AuthorizationID> of OCELOT.

CONNECT Examples

This SQL statement:

```
CONNECT TO 'some_server';
```

establishes an SQL-Connection to the SQL-server specified. The <Connection name> defaults to some_server; the SQL-session <AuthorizationID> is set to the DBMS's initial default <AuthorizationID>. This SQL statement:

```
CONNECT TO 'some_server' AS 'Connection_1';
```

establishes an SQL-Connection named Connection_1 to the SQL-server specified. The SQL-session <AuthorizationID> is set to the DBMS's initial default <AuthorizationID>. This SQL statement:

```
CONNECT TO 'some_server' USER 'bob';
```

establishes an SQL-Connection to the SQL-server specified. The <Connection name> defaults to some_server; the SQL-session <AuthorizationID> is set to bob. And this SQL statement:

```
CONNECT TO 'some_server' AS 'Connection_1' USER 'bob';
```

establishes an SQL-Connection named Connection_1 to the SQL-server specified. The SQL-session <AuthorizationID> is set to bob.

Executing the CONNECT statement has the effect that the SQL-Connection established becomes the current SQL-Connection and its associated SQL-session becomes the current SQL-session. The SQL-Connection and SQL-session that were current when you executed CONNECT (if any) become dormant with their context information preserved by the DBMS so that they can be properly restored later on. If the CONNECT statement fails, the current SQL-Connection and its associated SQL-session (if any) remain the current SQL-Connection and current SQL-session.

If CONNECT fails because your DBMS is unable to establish the SQL-Connection, you'll get the SQLSTATE error 08001 "connection exception-SQL-client unable to establish SQL-connection." If CONNECT fails because the SQL-server refuses to accept the SQL-Connection, you'll get the SQLSTATE error 08004 "connection exception-SQL-server rejected establishment of SQL-connection."

If you want to restrict your code to Core SQL, don't use the CONNECT statement.

SET CONNECTION Statement

[NON-PORTABLE] An SQL-compliant DBMS can either limit the number of concurrent SQL-Connections to one or it can support multiple concurrent SQL-Connections. [OCELOT Implementation] The OCELOT DBMS that comes with this book allows multiple concurrent SQL-Connections to be made; each begins a separate SQL-session for the <Cluster name> specified. Thus, OCELOT supports multi-user operations — one or more Users may connect to the same Cluster simultaneously — and OCELOT supports multi-tasking operations — the same user may connect to multiple Clusters simultaneously. Each such connection is a separate SQL-Connection (it must be identified by a unique <Connection name>) and is associated with a separate SQL-session.

The SET CONNECTION statement is used to select an SQL-Connection from all available SQL-Connections — it makes a dormant SQL-Connection current. As a consequence, any other SQL-Connection that was current then becomes dormant. The required syntax for the SET CONNECTION statement is as follows.

```
SET CONNECTION DEFAULT | <Connection name>
```

The SET CONNECTION statement activates a dormant SQL-Connection and makes it the current SQL-Connection. SET CONNECTION may not be executed during a transaction unless your DBMS supports transactions that affect multiple SQL-servers. The SQL statement:

```
SET CONNECTION DEFAULT;
```

will establish your DBMS's default SQL-Connection as the current SQL-Connection. If there is no current or dormant default SQL-Connection (that is, if CONNECT TO DEFAULT; wasn't previously issued during the SQL-session), SET CONNECTION will fail; your DBMS will return the SQLSTATE error 08003 "connection exception-connection does not exist."

The syntax SET CONNECTION <Connection name>; will establish the SQL-Connection specified as the current SQL-Connection. <Connection name> must be a <simple value specification> — e.g., a <character string literal>, <host parameter name>, or <SQL parameter reference> — whose value identifies the current, or a dormant, SQL-Connection. If <Connection name> does not evaluate to either the current or a dormant SQL-Connection, SET CONNECTION will fail; your DBMS will return the SQLSTATE error 08003 "connection exception-connection does not exist." For example, this SQL statement:

```
SET CONNECTION 'connection_2';
```

makes the SQL-Connection called connection_2 the current SQL-Connection and puts the previous (if any) SQL-Connection into a dormant state. If your DBMS is unable to activate connection_2, SET CONNECTION will fail and your DBMS will return the SQLSTATE error 08006 "connection exception-connection failure."

If you want to restrict your code to Core SQL, don't use the SET CONNECTION statement.

DISCONNECT Statement

An SQL-Connection can be closed whether it is the current SQL-Connection or a dormant SQL-Connection, but may not be closed while a transaction is on-going for its associated SQL-session. The required syntax for the DISCONNECT statement is as follows.

```
DISCONNECT <Connection name> | DEFAULT | CURRENT | ALL
```

The DISCONNECT statement terminates an inactive SQL-Connection. DISCONNECT may not be executed during a transaction — if you attempt to terminate an SQL-Connection that is processing a transaction, DISCONNECT will fail; your DBMS will return the SQLSTATE error 25000 "invalid transaction state."

You can disconnect a specific SQL-Connection by naming it (with DISCONNECT <Connection name>;), you can disconnect your DBMS's default SQL-Connection (with DISCONNECT DEFAULT;), you can disconnect the current SQL-Connection (with DISCONNECT CURRENT;) or you can disconnect the current and all dormant SQL-Connections at once (with DISCONNECT

ALL;). For example, this SQL statement closes an inactive SQL-Connection called connection_1, whether it is current or dormant;

```
DISCONNECT 'connection_1';
```

As usual, <Connection name> must be a <simple value specification> — e.g., a <character string literal>, <host parameter name>, or <SQL parameter reference> — whose value identifies the current, or a dormant, SQL-Connection. (If <Connection name> is not the name of either the current SQL-Connection or a dormant SQL-Connection, DISCONNECT will fail; your DBMS will return the SQLSTATE error 08003 "connection exception-connection does not exist.") If <Connection name> names the current SQL-Connection and DISCONNECT executes successfully, there will no longer be a current SQL-Connection until another CONNECT statement or SET CONNECTION statement establishes one. This SQL statement:

```
DISCONNECT DEFAULT;
```

terminates the DBMS's default SQL-Connection whether it is current or dormant. (If the DBMS's default SQL-Connection is neither the current SQL-Connection nor a dormant SQL-Connection, DISCONNECT will fail; your DBMS will return the SQLSTATE error 08003 "connection exception-connection does not exist.") If the default SQL-Connection is the current SQL-Connection and DISCONNECT executes successfully, there will no longer be a current SQL-Connection until another CONNECT statement or SET CONNECTION statement establishes one. This SQL statement:

```
DISCONNECT CURRENT;
```

terminates the current SQL-Connection. If there is no current SQL-Connection, DISCONNECT will fail; your DBMS will return the SQLSTATE error 08003 "connection exception-connection does not exist." If DISCONNECT executes successfully, there will no longer be a current SQL-Connection until another CONNECT statement or SET CONNECTION statement establishes one. This SQL statement:

```
DISCONNECT ALL;
```

closes the current, and all dormant, SQL-Connections. If there are no current or dormant SQL-Connections, DISCONNECT will fail; your DBMS will return the SQLSTATE error 08003 "connection exception-connection does not exist." If DISCONNECT executes successfully, there will no longer be any SQL-Connection (current or dormant) until another CONNECT statement or SET CONNECTION statement establishes one.

Any errors other than SQLSTATE 08003 or SQLSTATE 25000 that are detected by your DBMS while DISCONNECT is being executed will not cause DISCONNECT to fail. Instead, DISCONNECT will execute successfully and your DBMS will return the SQLSTATE warning 01002 "warning-disconnect error."

The SQL Standard suggests that DISCONNECT should be automatic when an SQL-session ends — but lets the DBMS decide when this has occurred. Recommendation: To be absolutely sure of correct results, always end your SQL-sessions with the explicit DISCONNECT statement:

```
DISCONNECT ALL;
```

If you want to restrict your code to Core SQL, don't use the DISCONNECT statement.

SQL-Session Management

SQL provides four statements that help you manage your SQL-session. Each one lets you specify a value for one or more SQL-session characteristics. The SQL-session management statements are SET SESSION CHARACTERISTICS, SET SESSION AUTHORIZATION, SET ROLE, and SET TIME ZONE.

SET SESSION CHARACTERISTICS Statement

The SET SESSION CHARACTERISTICS statement sets the value of one or more transaction characteristics for the current SQL-session. The required syntax for the SET SESSION CHARACTERISTICS statement is as follows.

```
SET SESSION CHARACTERISTICS AS
   <transaction mode> [ {,<transaction mode>}... ]

   <transaction mode> ::=
   <isolation level> |
   <transaction access mode> |
   <diagnostics size>
```

You can set the same characteristics for all the transactions in an entire SQL-session as you can for a single transaction. Each of the transaction characteristics — <isolation level>, <transaction access mode>, and <diagnostics size> — works the same and has the same options as those we discussed for the SET TRANSACTION statement in Chapter 37. The values you specify for any transaction characteristic in a SET SESSION CHARACTERISTICS statement are enduring values — should you cause the current SQL-session to go dormant and then reactivate it later, your DBMS will reset each characteristic to the value you specified the last time you issued SET SESSION CHARACTERISTICS for that SQL-session. Here's an example:

```
SET SESSION CHARACTERISTICS AS
   READ WRITE
   ISOLATION LEVEL REPEATABLE READ
   DIAGNOSTICS SIZE 5
```

If you want to restrict your code to Core SQL, don't use the SET SESSION CHARACTERISTICS statement.

SET SESSION AUTHORIZATION Statement

The SET SESSION AUTHORIZATION statement sets the session user <AuthorizationID> for the current SQL-session. The required syntax for the SET SESSION AUTHORIZATION statement is as follows.

```
SET SESSION AUTHORIZATION <value specification>
```

When you start an SQL-session, your DBMS sets the value of the session user <AuthorizationID> for the SQL-session to the <AuthorizationID> specified with the CONNECT statement. The session user <AuthorizationID> is the value returned by the SESSION_USER function and is

usually also the value returned by the CURRENT_USER (or USER) function. Your DBMS uses the session <AuthorizationID> as a default <AuthorizationID> in cases where no explicit <AuthorizationID> overrides it — for example, whenever you run a Module that wasn't defined with an explicit AUTHORIZATION clause, your DBMS assumes the owner of the Module is the SQL-session <AuthorizationID>. The owner of any temporary Tables defined for the SQL-session is the SQL-session <AuthorizationID>.

[NON-PORTABLE] SET SESSION AUTHORIZATION may always be executed at the start of an SQL-session. Whether you can use the SET SESSION AUTHORIZATION statement at any other time is non-standard because the SQL Standard requires implementors to define whether the SQL-session <AuthorizationID> may be changed once an SQL-session has begun. [OCELOT Implementation] The OCELOT DBMS that comes with this book allows the SQL-session <AuthorizationID> to be changed at any time (except during a transaction).

You can change the value of the SQL-session <AuthorizationID> with the SET SESSION AUTHORIZATION statement; simply issue SET SESSION AUTHORIZATION followed by a <character string literal>, a character string <host parameter name> (with optional indicator), a character string <SQL parameter reference>, or a user function (either CURRENT_ROLE, CURRENT_USER, SESSION_USER, SYSTEM_USER, or USER). Whichever you use, the value represented by the <value specification> must be a valid user <AuthorizationID> — if it isn't, SET SESSION AUTHORIZATION will fail; your DBMS will return the SQLSTATE error 28000 "invalid authorization specification."

SET SESSION AUTHORIZATION can only be issued outside of a transaction. If you try to execute the statement and a transaction is currently active, SET SESSION AUTHORIZATION will fail; your DBMS will return the SQLSTATE error 25001 "invalid transaction state-active SQL-transaction."

For an example of SET SESSION AUTHORIZATION, assume that the session user <AuthorizationID> for your SQL-session is bob and you'd like to switch it to sam. Here's three different ways to do it:

```
SET SESSION AUTHORIZATION 'sam';

SET SESSION AUTHORIZATION :char_variable;
   -- assume the value of the host variable "char_variable" is SAM

SET SESSION AUTHORIZATION CURRENT_USER;
   -- assume the value of CURRENT_USER is SAM
```

If you want to restrict your code to Core SQL, don't use the SET SESSION AUTHORIZATION statement.

SET ROLE Statement

The SET ROLE statement sets the enabled Roles for the current SQL-session. The required syntax for the SET ROLE statement is as follows.

```
SET ROLE <value specification> | NONE
```

When you start an SQL-session, your DBMS sets the value of the current Role <AuthorizationID> for the SQL-session to the <AuthorizationID> specified with the CONNECT statement

(or to NULL, if the CONNECT statement doesn't provide a <Role name>). The current Role <AuthorizationID> is the value returned by the CURRENT_ROLE function. Either one of CURRENT_USER or CURRENT_ROLE may be NULL at any time, but they may not both be NULL at the same time — the non-null identifier is the SQL-session's current <AuthorizationID>. That is, if CURRENT_ROLE is set to some <Role name>, then CURRENT_USER must be NULL and your DBMS will use the current Role for Privilege checking before processing any SQL statements in the SQL-session.

You can change the value of CURRENT_ROLE with the SET ROLE statement; simply issue SET ROLE followed by a <character string literal>, a character string <host parameter name> (with optional indicator), a character string <SQL parameter reference>, or a user function (either CURRENT_ROLE, SESSION_USER, SYSTEM_USER, or USER). Whichever you use, the value represented by the <value specification> must be a valid <Role name> and that name must identify a Role that has been granted either to PUBLIC or to the SQL-session <AuthorizationID> — if it isn't, SET ROLE will fail; your DBMS will return the SQLSTATE error 0P000 "invalid role specification." You can also change the value of CURRENT_ROLE to NULL by issuing SET ROLE followed by the <keyword> NONE.

SET ROLE can only be issued outside of a transaction. If you try to execute the statement and a transaction is currently active, SET ROLE will fail; your DBMS will return the SQLSTATE error 25001 "invalid transaction state-active SQL-transaction."

For an example of SET ROLE, assume that the current Role for your SQL-session is NULL and you'd like to switch it to Teller_Role. Here's two different ways to do it:

```
SET ROLE 'Teller_Role';

SET ROLE :char_variable;
   -- assume the value of the host variable "char_variable" is TELLER_ROLE
```

If you want to restrict your code to Core SQL, don't use the SET ROLE statement.

SET TIME ZONE Statement

[NON-PORTABLE] An SQL-session always begins with an initial default time zone offset that is non-standard because the SQL Standard requires implementors to define their own initial default time zone offset. [OCELOT Implementation] The OCELOT DBMS that comes with this book has an initial default time zone that represents UTC — its default time zone offset is INTERVAL +'00:00' HOUR TO MINUTE.

The SQL-session default time zone offset is used to specify the related time zone for all times and timestamps that don't include an explicit <time zone interval>. You can use the SET TIME ZONE statement to change the default time zone offset for the current SQL-session. The required syntax for the SET TIME ZONE statement is as follows.

```
SET TIME ZONE LOCAL | interval_expression
```

The SET TIME ZONE statement changes the current SQL-session's default time zone offset. It has two possible arguments: the <keyword> LOCAL or an expression that evaluates to some non-null INTERVAL HOUR TO MINUTE value between INTERVAL -'12:59' HOUR TO MINUTE and INTERVAL +'13:00' HOUR TO MINUTE. (If you specify an interval that is NULL or an interval that

falls outside the proper range, SET TIME ZONE will fail; your DBMS will return the SQLSTATE error 22009 "data exception-invalid time zone displacement value.")

SET TIME ZONE can only be issued outside of a transaction. If you try to execute the statement and a transaction is currently active, SET TIME ZONE will fail; your DBMS will return the SQLSTATE error 25001 "invalid transaction state-active SQL-transaction."

The effect of this SQL statement:

```
SET TIME ZONE LOCAL;
```

is to set the time zone offset for the current SQL-session to your DBMS's initial default time zone offset.

The SQL syntax SET TIME ZONE interval_expression is used to set the time zone offset for the current SQL-session to the value that results when interval_expression is evaluated. For example, this SQL statement:

```
SET TIME ZONE INTERVAL -'03:00' HOUR TO MINUTE;
```

uses the <interval literal> INTERVAL -'03:00' HOUR TO MINUTE to set the time zone offset for the current SQL-session to minus three hours, i.e., UTC time plus 3 hours equals local time.

If you want to restrict your code to Core SQL, don't use the SET TIME ZONE statement.