NAME

close −close cursor

SYNTAX

close *cursor-name*

DESCRIPTION

The **close** statement terminates the specified cursor. The access method automatically releases resources associated with the closed cursor. The **commit** and **rollback** statements, and **prepare** statement automatically close all cursors.

Once you have closed a cursor, you cannot issue any more **fetch** statements against that cursor unless you explicitly re-open it with another **open** statement. Records selected for that cursor’s active set are no longer available to your program. The active set of the cursor is said to be ‘‘undefined.’’

ARGUMENTS

cursor-name Identifies the cursor you want to close.

EXAMPLE

The following example declares a cursor, opens it, accesses records in its active set, and then closes the cursor:

```
. sql_8a.epas
       program mapper (input_output);
       exec sql
         begin declare section;
       exec sql
         end declare section;

       var
         statecode  :  array [1..2] of char;
         cityname  :  array [1..15] of char;

       begin

       exec sql
         declare bigcities cursor for
           select city, state from cities
           where population > 1000000;

       exec sql
         open bigcities;
```

```
exec sql
  fetch bigcities into :cityname, :statecode;

writeln (’ ’);
while (sqlcode = 0) do
begin
  writeln (cityname, ’ is in ’, statecode);
  exec sql
    fetch bigcities into :cityname, :statecode;
end;

exec sql
  close bigcities;
exec sql
  rollback release;
end.
```

SEE ALSO

See the entries in this chapter for:

- **open**
- **commit**
- **rollback**

DIAGNOSTICS

The access method returns errors if:

- You fetch beyond the last record of an active set, automatically closes the cursor and returns an end-of-file error.
- You try to close a cursor that has not been opened, returns an error.

The following values may be returned to SQLCODE:

- *SQLCODE < 0* indicates that the statement did not complete.
- *SQLCODE = 0* indicates success.
- *SQLCODE > 0 and < 100* indicates an informational message or warning.

See Chapter 6 for a discussion of error handling.

NAME

commit −write changes to database

SYNTAX

commit [**work**] [**release**]

DESCRIPTION

The **commit** statement:

- Ends the current transaction
- Makes the transaction’s changes visible to other users
- Closes open cursors
- Does not affect the contents of host variables

ARGUMENTS

work An optional noiseword.

release Breaks your program’s connection to the attached database, thus making system resources available to other users.

EXAMPLE

The following program illustrates the use of multiple cursors in a single transaction, terminated by a single **commit** that makes all changes permanent:

```
. sql_110a.epas
       program update_census (input_output);
       exec sql
         include sqlca;
       var
         newcity, oldcity  :  array [1..15] of char;
         state      :  array [1..2] of char;
         first      :  boolean;
         option      :  char;

       begin
       write (’Enter the city name that”s changing: ’);
       readln (oldcity);
       write (’Enter the new city name: ’);
       readln (newcity);
       writeln (’Changing ’, oldcity, ’ to ’, newcity, ’ in all relations’);
```

```
exec sql
  declare cities_cursor cursor for
  select state from cities
  where city = :oldcity
  for update of city;
exec sql
  declare tourism_cursor cursor for
  select state from tourism
  where city = :oldcity
  for update of city;
exec sql
  declare ski_areas_cursor cursor for
  select state from  ski_areas
  where city = :oldcity
  for update of city;
exec sql
  open ski_areas_cursor;
exec sql
  open tourism_cursor;
exec sql
  open cities_cursor;

first  := true;
while sqlcode = 0 do begin
  if not first then
  begin
    write (’Change ’, oldcity, state, ’ in cities? ’);
    readln (option);
    if (option = ’y’) then
      exec sql update cities
        set city = :newcity
        where current of cities_cursor;
  end;
  exec sql
    fetch cities_cursor into :state;
  first := false;
end;

sqlcode := 0;
first := true;
while sqlcode = 0 do begin
  if not first then
  begin
```

```
    write (’Change ’, oldcity, state, ’ in tourism? ’);
    readln (option);
    if (option = ’y’) then
      exec sql
        update tourism
        set city = :newcity
        where current of tourism_cursor;
  end;
  exec sql
    fetch tourism_cursor into :state;
  first := false;
end;

sqlcode := 0;
first := true;
while sqlcode = 0 do begin
  if not first then
  begin
    write (’Change ’, oldcity, state, ’ in ski areas? ’);
    readln (option);
    if (option = ’y’) then
      exec sql
        update ski_areas
        set city = :newcity
        where current of ski_areas_cursor;
  end;
   exec sql
    fetch ski_areas_cursor into :state;
  first := false;
end;
exec sql
  close ski_areas_cursor;
exec sql
  close tourism_cursor;
exec sql
  close cities_cursor;
 exec sql
  commit release;

end.
```

SEE ALSO

See the entry in this chapter for **rollback**.

DIAGNOSTICS

The following values may be returned to SQLCODE:

- *SQLCODE < 0* indicates that the statement did not complete.
- *SQLCODE = 0* indicates success.
- *SQLCODE > 0 and < 100* indicates an informational message or warning.
- *SQLCODE = 100* indicates the end of the active set.

See Chapter 6 for a discussion of error handling.

NAME

declare cursor −define cursor

SYNTAX

declare *cursor-name* **cursor for** *select-statement*
[**for update of** *database-field-commalist*]
[**order by** *sort-key-commalist*]

sort-key ::= *field-reference* [**asc** | **desc**]

field-reference ::= { *database-field* | *integer* }

DESCRIPTION

The **declare cursor** declaration defines a cursor by associating a name with the active set of records determined by a **select** statement.

ARGUMENTS

cursor-name Provides a name for the cursor you are declaring.

select-statement A **select** statement that specifies search conditions to determine the active set of the cursor.

order by Specifies the order in which the retrieved records are to be delivered to the program. You can sort records by named fields in the source relation(s) or by an *integer* that references by position one of the fields in the **select** statement.

for update Indicates that your program may update one or more fields of records in the active set. Standard restricts you to updating only the listed fields; however, does not enforce this restriction.

EXAMPLE

The following example declares a cursor a search condition and a sorting clause:

```
. sql_116a.epas
       program sql (input, output);
       exec sql
         include sqlca;

       var
         statecode  :  array [1..2] of char;
         cityname  :  array [1..15] of char;
         min_pop    :  integer32;
         option    :  char;
       begin
```

```
min_pop := 100;

(* the crude way *)

exec sql
  delete from cities
  where population < :min_pop;

exec sql
  rollback;

(* with finesse *)

exec sql
  declare small_cities cursor for
    select city, state
    from cities
    where population < :min_pop;
exec sql
  open small_cities;
exec sql
  fetch small_cities into :cityname, :statecode;

while sqlcode = 0 do
begin
  write (’Eliminate ’, cityname, ’ ’, statecode, ’? ’);
  readln (option);
  if (option = ’Y’) or (option = ’y’) then
  exec sql
    delete from cities
    where current of small_cities;
  exec sql
    fetch small_cities into :cityname, :statecode;
end;

exec sql
  close small_cities;
exec sql
  rollback release;

end.
```

The following example declares a cursor for two relations:

```
. sql_31a.epas
       program sql (input, output);
       exec sql
         include sqlca;

       var
         city, lat, long  :  array [1..15] of char;
         state    :  array [1..20] of char;

       begin

       exec sql
         declare city_state_join cursor for
           select c.city, s.state_name, c.latitude, c.longitude
           from cities c, states s where c.state = s.state
           order by s.state, c.city;

       exec sql
         open city_state_join;
       exec sql
         fetch city_state_join into :city, :state, :lat, :long;

       while (sqlcode = 0) do begin
         writeln (city, state, lat, long);
         exec sql
           fetch city_state_join into :city, :state, :lat, :long;

       end;
       exec sql
         rollback release;

       end.
```

The following program declares a cursor with the union of three relations.

```
. sql_31c.epas
       program sql (input, output);
       exec sql
         include sqlca;

       var
         city  :  array [1..25] of char;
         state  :  array [1..2] of char;
```

```
begin

exec sql
  declare  all_cities cursor for
    select city, state from cities
    union
    select city, state from ski_areas
    union
    select capitol, state from states
    order by 2, 1;
exec sql
  open all_cities;
exec sql
  fetch all_cities into :city, :state;

while (sqlcode = 0) do begin
  writeln (city, state);
  exec sql
    fetch all_cities into :city, :state;

end;
exec sql
  rollback release;

end.
```

SEE ALSO

See the entry for **select** in this chapter.

DIAGNOSTICS

The following values may be returned to SQLCODE:

- *SQLCODE < 0* indicates that the statement did not complete.
- *SQLCODE = 0* indicates success.
- *SQLCODE > 0 and < 100* indicates an informational message or warning.

See Chapter 6 for a discussion of error handling.

NAME

delete −erase record

SYNTAX

delete from *relation-name* [*alias*]
[**where** *predicate* | **where current of** *cursor-name*]

DESCRIPTION

The **delete** statement erases one or more records in a relation or in the active set of a cursor:

If you do not provide a search condition (**where**...), all records in the specified relation are deleted. Be very careful with this option.

ARGUMENTS

relation-name Specifies the relation from which a record is to be deleted.

alias Qualifies field references with an identifier that indicates the source relation. The *alias* can be useful if the *predicate* references fields from different relations.

The *alias* can contain up to 31 alphanumeric characters, dollar signs ($), and underscores (_). However, it must start with an alphabetic character (A—Z, a—z). Except for C programs, **gpre** is not sensitive to the case of the alias. For example, it treats **B** and **b** as the same character. For C programs, you can control the case sensitivity of the alias with the **either_case** switch when you preprocess your program.

where *predicate* Determines the record to be deleted.

where current Specifies that the current record of the active set is to be deleted. This form of **delete** must follow:

- The declaration of the cursor with a **declare cursor** statement
- The opening of that cursor with an **open** statement
- The retrieval of a record from the active set of that cursor with a **fetch** statement

EXAMPLES

The following statement erases the entire relation named VILLAGES (which does not exist in the sample database):

```
. tcs:
       .  delete_2.epas in a manner of speaking
       exec sql delete from villages;
```

The following program deletes all records from CITIES with a population less than that of the host variable MIN_POP:

```
. sql_116a.epas
       program sql (input, output);
       exec sql
         include sqlca;

       var
         statecode  :  array [1..2] of char;
         cityname  :  array [1..15] of char;
         min_pop    :  integer32;
         option    :  char;
       begin

       min_pop := 100;

       (* the crude way *)

       exec sql
         delete from cities
         where population < :min_pop;

       exec sql
         rollback;

       (* with finesse *)

       exec sql
         declare small_cities cursor for
           select city, state
           from cities
           where population < :min_pop;
       exec sql
         open small_cities;
       exec sql
         fetch small_cities into :cityname, :statecode;

       while sqlcode = 0 do
       begin
         write (’Eliminate ’, cityname, ’ ’, statecode, ’? ’);
         readln (option);
         if (option = ’Y’) or (option = ’y’) then
         exec sql
```

```
     delete from cities
     where current of small_cities;
   exec sql
     fetch small_cities into :cityname, :statecode;
 end;

 exec sql
   close small_cities;
 exec sql
   rollback release;

 end.
```

SEE ALSO

See the entries in this chapter for:

- *predicate*
- **declare cursor**
- **open**
- **fetch**
- **select**

DIAGNOSTICS

The following values may be returned to SQLCODE:

- *SQLCODE < 0* indicates that the statement did not complete.
- *SQLCODE = 0* indicates success.
- *SQLCODE > 0 and < 100* indicates an informational message or warning.

See Chapter 6 for a discussion of error handling.

NAME

fetch −advance cursor

SYNTAX

fetch *cursor-name* [**into** *host-item-commalist*]

host-item ::= :*host-variable*

DESCRIPTION

The **fetch** statement advances the position of the cursor to the next record of the active set.

If the **fetch** statement immediately follows an **open** statement, the cursor is set before the first record in that cursor. The **fetch** statement advances the cursor to the first record.

If you try to fetch beyond the last record in the active set, automatically closes the cursor and returns an end-of-file message.

Once the **fetch** statement has advanced the cursor, it writes the fields of that record into the listed host variables. Because the **select** substatement in the **declare cursor** statement explicitly lists database field names, you must make sure that the host variables correspond exactly to the order of declaration in the cursor, the datatypes, and lengths of the database fields. For example, if you want to fetch a database field of 10 characters that appears as the third item in the cursor declaration, make sure that the host variable:

- Is also a text field with a minimum of 10 characters
- Appears in the third position of the host variable list

If you want to update or delete a record in a cursor's active set, you must first fetch it. You can then use the **update** statement to modify one or more of its field values, or use the **delete** statement to erase it.

If you want to loop through the records selected by the cursor, enclose the **fetch** statement in a host language looping construct.

ARGUMENTS

cursor-name Specifies the open cursor from which you want to fetch records.

host-item Specifies a host language variable into which fields from records in the active set of the cursor will be fetched. The **into** list is not required if the **fetch** gets records to be deleted or updated; however, if you display the record before you delete or update it, you need the **into** list.

EXAMPLE

The following example declares a cursor, opens it, accesses records in its active set, and then closes the cursor:

```
 . sql_8a.epas
         program mapper (input_output);
```

```
exec sql
  begin declare section;
exec sql
  end declare section;

var
  statecode  :  array [1..2] of char;
  cityname  :  array [1..15] of char;

begin

exec sql
  declare bigcities cursor for
    select city, state from cities
    where population > 1000000;

exec sql
  open bigcities;
exec sql
  fetch bigcities into :cityname, :statecode;

writeln (’ ’);
while (sqlcode = 0) do
begin
  writeln (cityname, ’ is in ’, statecode);
  exec sql
    fetch bigcities into :cityname, :statecode;
end;

exec sql
  close bigcities;
exec sql
  rollback release;
end.
```

The following program extract uses a **fetch** statement in a loop that modifies records:

```
. sql_120a.epas
       program popupdate (input_output);
       exec sql
         begin declare section;
       exec sql
         end declare section;
```

```
var
  statecode, st  :   array [1..2] of char;
  cityname  :  array [1..15] of char;
  multiplier  :  integer32;
  pop, new_pop  :  integer32;

begin

write (’Enter state with population needing adjustment: ’);
readln (statecode);

exec sql
  declare pop_mod cursor for
    select city, state, population from cities
    where state = :statecode
    for update of population;

exec sql
  open pop_mod;
exec sql
  fetch pop_mod into :cityname, :st, :pop;

writeln (’ ’);
while (sqlcode = 0) do
begin
  write (’Change for ’, cityname,
    st, ’ (5 => 5% bigger; -5 => 5% smaller): ’);
  readln (multiplier);
        new_pop := trunc (pop * (multiplier + 100) / 100);
  writeln (’ old population: ’, pop, ’ new population: ’, new_pop);
  exec sql
    update cities
      set population = :new_pop
      where current of pop_mod;
  exec sql
    fetch pop_mod into :cityname, :st, :pop;
end;

exec sql
  close pop_mod;
exec sql
  rollback release;
end.
```

SEE ALSO

See the entries in this chapter for:

- **open**
- **declare cursor**
- **select**
- **update**
- **delete**
- **whenever**

DIAGNOSTICS

The following values may be returned to SQLCODE:

- *SQLCODE < 0* indicates that the statement did not complete.
- *SQLCODE = 0* indicates success.
- *SQLCODE > 0 and < 100* indicates an informational message or warning.
- *SQLCODE = 100* indicates the end of the active set.

See Chapter 6 for a discussion of error handling.

NAME

insert −store a record

SYNTAX

insert into *relation-name* [*database-field-commalist*]
{ **values** *insert-item-commalist* | *select-statement* }

insert-item ::= { *constant* | *host-variable* | **null** }

DESCRIPTION

The **insert** statement stores a new record into the specified relation.

You can assign field values by inserting values, by picking up values from an existing record, or by a combination of both.

ARGUMENTS

relation-name Specifies the relation into which you want to store a new record.

database-field Lists the field in *relation-name* for which you are providing a value.

by itself does not support manipulation of the blob datatype. You can store a null value for a blob field, but you must use or **gds** calls if you want to do anything else with blobs.

If the field you are assigning is a date, you cannot handle the field directly with Instead, you must use date handling functions such as **gds_$encode_date** and **gds_$decode_date** to convert your external date representation to a host variable in the date format (that is, an array of two 32-bit integers). Then use the assignment to assign the host variable to the database field.

NOTE

The database field list is optional. If it is omitted, values are assigned to all the fields in the relation in their normal order. Leaving out the field list is *not* recommended because changes to the relation, such as adding or reordering fields, will cause the assignment list to change without warning when the program is next precompiled with **gpre**.

insert-item Provides a value for *database-field*. The value can be a constant, host variable, or **null**.

select-statement Specifies that the values for the new record are to come from the record identified by a **select** statement.

EXAMPLES

The following program stores a record, assigning quoted constants for field values:

```
. sql_125a.epas
       program sql (input, output);
       exec sql
         include sqlca;

       begin

       exec sql
         insert into river_states
         (river, state)
         values ('Croton', 'NY');

       exec sql
         rollback release;

       end.
```

The following statement stores a new record into STATES using host variables and **null** as sources for values:

```
. sql_123b.epas
       program sql (input, output);
       exec sql
         include sqlca;

       var
         state    :  array [1..2] of char;
         state_name  :  array [1..20] of char;
         capitol    :  array [1..15] of char;
         date    :  gds_$quad;
         date_array  :  gds_$tm;

       begin

       date_array.tm_sec := 0;
       date_array.tm_min := 0;
       date_array.tm_hour := 0;
       date_array.tm_mday := 1;
       date_array.tm_mon := 1;
       date_array.tm_year := 90;
       date_array.tm_wday := 0;
```

```
        date_array.tm_yday := 0;
        date_array.tm_isdst := 0;

        gds_$encode_date (date_array, date);
        state := ’GU’;
        state_name := ’Guam’;
        capitol := ’Agana’;
        exec sql
          insert into states
          (state, state_name, area, capitol, statehood)
          values (:state, :state_name, null, :capitol, :date);


        exec sql
          commit release;

        end.
```

The following program stores a new record using values from an existing record and the value of a host variable for assignments:

```
. sql_123c.epas
        program sql (input, output);
        exec sql
          include sqlca;

        var
          villeancienne  :  array [1..15] of char;
          villenouvelle  :  array [1..15] of char;


        begin

        write (’Enter city to clone: ’);
        readln (villeancienne);
        write (’Enter new name for city: ’);
        readln (villenouvelle);

        exec sql insert into cities (city, state, population,
            altitude, latitude_degrees, latitude_minutes,
            latitude_compass, longitude_degrees, longitude_minutes,
            longitude_compass)
          select :villenouvelle, state, population,
            altitude, latitude_degrees, latitude_minutes,
            latitude_compass, longitude_degrees, longitude_minutes,
            longitude_compass
          from cities where city = :villeancienne;
        end.
```

The following program uses the non-recommended form of the **insert** statement, in which the database field list is omitted:

```
. sql_123d.epas
       program sql (input, output);
       exec sql
         include sqlca;

       var
         state    :  array [1..2] of char;
         state_name  :  array [1..20] of char;
         capitol    :  array [1..15] of char;

       begin

       state := ’GU’;
       state_name := ’Guam’;
       capitol := ’Agana’;
       exec sql
         insert into states
         values (:state, :state_name, null, null, :capitol);
       exec sql
         commit release;

       end.
```

SEE ALSO

See the entry for **select** in this chapter.

DIAGNOSTICS

The following values may be returned to SQLCODE:

- *SQLCODE < 0* indicates that the statement did not complete.
- *SQLCODE = 0* indicates success.
- *SQLCODE > 0 and < 100* indicates an informational message or warning.

See Chapter 6 for a discussion of error handling.

NAME

open −activate cursor

SYNTAX

open *cursor-name*

DESCRIPTION

The **open** statement activates a cursor. This statement causes the access method to evaluate the search conditions associated with the specified cursor. Once the access method has determined the set of records that satisfies the query, it activates the cursor and makes the selected records the ‘‘active set’’ of that cursor.

The access method then places the cursor itself before the first record in the active set. If you want to retrieve or update records in that set, use the **fetch** statement. Once you open the cursor, the first **fetch** statement operates on the very first record in the active set. Subsequent **fetch** statements advance the cursor through the results table associated with that cursor.

The access method does not re-examine the host variables or values passed to the search conditions until you close the cursor and re-open it. Changes you make to their values are not reflected in the active set until you close and re-open the cursor. If someone else accesses the database after you open a cursor, makes changes, and commits them, the active set may be different the next time you open that cursor if you commit your transaction.

If you need a stable active set, use the **consistency** option of the **start_transaction** statement.

ARGUMENTS

cursor-name Specifies the declared cursor you want to access.

EXAMPLE

The following example declares a cursor, opens it, accesses records in its active set, and then closes the cursor:

```
. sql_8a.epas
       program mapper (input_output);
       exec sql
         begin declare section;
       exec sql
         end declare section;

       var
         statecode  :  array [1..2] of char;
         cityname  :  array [1..15] of char;

       begin
```

```
exec sql
  declare bigcities cursor for
    select city, state from cities
    where population > 1000000;

exec sql
  open bigcities;
exec sql
  fetch bigcities into :cityname, :statecode;

writeln (’ ’);
while (sqlcode = 0) do
begin
  writeln (cityname, ’ is in ’, statecode);
  exec sql
    fetch bigcities into :cityname, :statecode;
end;

exec sql
  close bigcities;
exec sql
  rollback release;
end.
```

SEE ALSO

See the entries in this chapter for:

- **declare cursor**
- **fetch**
- **close**
- **commit**
- **rollback**
- **whenever**

DIAGNOSTICS

The following values may be returned to SQLCODE:

- *SQLCODE < 0* indicates that the statement did not complete.
- *SQLCODE = 0* indicates success.
- *SQLCODE > 0 and < 100* indicates an informational message or warning.

See Chapter 6 for a discussion of error handling.

NAME

predicate −specify Boolean expression

SYNTAX

predicate ::= { *condition* | *condition* **and** *predicate* |
condition **or** *predicate* | **not** *predicate* }

condition ::= { *compare-condition* | *between-condition* |
like-condition | *in-condition* | *exists-condition* | *(predicate)* }

DESCRIPTION

The *predicate* clause is used to select the records to be affected by the statement. It is used in the **where** clause of the **delete** and **update** statements and in the *select-expression*.

ARGUMENTS

compare-condition The *compare-condition* describes the characteristics of a single scalar expression (for example, a missing or null value) or the relationship between two scalar expressions (for example, *x* is greater than *y*).

Syntax: compare-condition of Predicate

{ *scalar-expression* *comparison-operator* *scalar-expression* |
scalar-expression *comparison-operator* *(column-select-expression)* |
scalar-expression **is** [**not**] **null** }

comparison-operator ::= { = | ˆ= | < | ˆ< | <= | > | ˆ> | >= }

column-select-expression ::=
select [**distinct**] *scalar-expression* *from-clause* [*where-clause*]

between-condition The *between-condition* specifies an inclusive range of values to match.

Format: between-condition of Predicate

database-field [**not**] **between** *scalar-expression-1*
and *scalar-expression-2*

like-condition Matches a string with the whole or part of a field value. The test is case-sensitive.

Format: like-condition of Predicate

database-field [**not**] **like** *scalar-expression*

The *scalar-expression* usually represents an alphanumeric literal, and can contain wildcard characters. Wildcard characters are:

- The underscore, _, that matches a single character.
- The percent sign, %, that matches any sequence of characters, including none. You should begin and end wildcard searches with the percent sign so that you match leading or trailing blanks.

in-condition Lists a set of scalar expressions as possible values.

Format: in-condition of Predicate

scalar-expression [**not**] **in** (*set-of-scalars*)

set-of-scalars ::= { *constant-commalist* | *column-select-expression* }

column-select-expression ::=
select [**distinct**] *scalar-expression* *from-clause* [*where-clause*]

exists-condition Tests for the existence of at least one qualifying record identified by the **select** subquery. Because the *exists-condition* uses the parenthesized **select** statement only to retrieve a record for comparison purposes, it requires only wildcard (*) field selection.

A predicate containing an *exists-condition* is true if the set of records specified by *select-expression* includes at least one record. If you add **not**, the predicate is true if there are *no* records that satisfy the subquery.

Format: exists-condition of Predicate

[**not**] **exists** (**select** * *where-clause*)

EXAMPLES

The following cursor retrieves all fields from CITIES records for which the POPULATION field is not missing:

```
. sql_130a.epas
        exec sql
          declare inhabited cursor for
```

```
            select city, state, population
            from cities
            where population is not null;
```

The following cursor retrieves the CITY and STATE fields from cities with populations between 100000 and 125000:

```
. sql_130a.epas
      exec sql
        declare midsized_cities cursor for
          select city, state
          from cities
          where population between 100000 and 125000;
```

The following cursor retrieves the CAPITOL and STATE from STATES records in which the CAPITOL field contains the string ‘‘ville’’ preceded or followed by any number of characters:

```
. sql_130a.epas
      exec sql
        declare ville cursor for
          select capitol, state
          from states
          where capitol like ’%ville%’;
```

SEE ALSO

See the entries in this chapter for:

- *select-expression*
- *scalar-expression*
- **delete**
- **update**

DIAGNOSTICS

See Chapter 6 for a discussion of error handling.

NAME

rollback −undo transaction

SYNTAX

rollback [**work**] [**release**]

DESCRIPTION

The **rollback** statement restores the database to its state prior to the current transaction. It also closes open cursors.

ARGUMENTS

work An optional noiseword.

release Breaks your program’s connection to the attached database, thus making system resources available to other users.

EXAMPLE

The following non-working code extract includes a **whenever** statement and the rollback routine to which it branches:

```
. sql_131a.epas
       program update_census (input_output);

       label
         error, warn, terminate;

       warn:
         (* since no warnings are defined, fall into error *)
       error:
         writeln (’Encountered SQL error code ’, sqlcode);
         writeln (’Expanded error listing: ’);
         gds_$print_status (gds_$status);
         if (sqlcode = -16) then
         begin
           exec sql
             rollback;
           work ();
         end
         else
           exec sql
             rollback release;
```

```
 terminate:
 end.
```

SEE ALSO

See the entries in this chapter for:

- **commit**
- **whenever**

DIAGNOSTICS

The following values may be returned to SQLCODE:

- *SQLCODE < 0* indicates that the statement did not complete.
- *SQLCODE = 0* indicates success.
- *SQLCODE > 0 and < 100* indicates an informational message or warning.

See Chapter 6 for a discussion of error handling.

NAME

scalar-expression −calculating value

SYNTAX

scalar-expression ::= [+ | -] *scalar-value* [*arithmetic-operator scalar-expression*]

scalar-value ::= { *field-expression* | *constant-expression* |
statistical-function | (*scalar-expression*) }

arithmetic-operator ::= { + | - | * | / }

DESCRIPTION

The *scalar-expression* is a symbol or string of symbols used in predicates to calculate a value. uses the result of the expression when executing the statement in which the expression appears.

You can add (+), subtract (-), multiply (*), and divide (/) scalar expressions. Arithmetic operations are evaluated in the normal order. You can use parentheses to change the order of evaluation.

ARGUMENTS

field-expression References a database field. The format of the *field-expression* follows:

Syntax: field-expression of Scalar Expression

[*database-handle.*] [*relation-name.* | *view-name.* | *alias.*]*database-field*

The optional *relation-name*, *view-name*, or *alias*, each followed by a required period (.), specifies the relation, view, or alias (synonym for a relation or view) in which the field is located. The alias is assigned to a relation or a view in a *select-expression*.

Use the optional *database-handle* only if you have declared a database handle with a **ready** statement.

constant-expression A string of ASCII digits interpreted as a number or as a string of ASCII characters. The format of the *constant-expression* follows:

Syntax: constant-expression Scalar Expression

{ *integer-string* | *decimal-string* | *float-string* | *ascii-string* }

Integer numeric strings are written as signed or unsigned decimal integers without decimal points. For example, the following are integers: *-14, 0, 9*, and *+47*.

Decimal numeric strings are written as signed or unsigned decimal integers with decimal points. For example, the following are decimal strings: *-14.3, 0.021, 9.0*, and *+47.9*.

Floating numeric strings are written in scientific notation (that is, *E-format*). A number in scientific notation consists of a decimal string mantissa, the letter *E*, and a signed integer exponent. For example, the following are floating numerics: *7.12E+7* and *7.12E-7*.

Character strings are written using ASCII printing characters enclosed in single (’) or double (") quotation marks. ASCII printing characters are:

- Uppercase alphabetic: *A—Z*
- Lowercase alphabetic: *a—z*
- Numerals: *0—9*
- Blank space and tab
- Special characters: ! @ # $ % ˆ & * ( ) _ - + = ‘ ˜ [ ] { } < > ; : ’ " \ | / ? . ,

statistical-function An expression that calculates a single value from the values of a field in a relation, view, or join. The format of the *statistical-function* follows:

Syntax: statistical-function Scalar Expression

{ **count (*)** |
function-name **(***scalar-expression***)** |
function-name **(distinct)** *field-expression* }

function-name ::= {\ **count** | **sum** | **avg** | **max** | **min** }

Supported statistical functions are:

- **count (*)** returns the number of records in a relation and automatically eliminates duplicates; **distinct** is not needed.

  If you are programming in Pascal, put a space between the open parenthesis and the asterisk. Because Pascal uses the sequence (* for comments, failure to leave a space will result in a compilation error.

- **count** returns the number of values for the field. You must specify **distinct**.
- **sum** returns the sum of values for a numeric field in all qualifying records.
- **avg** returns the average value for a numeric field in all qualifying records.
- **max** returns the largest value for the field.
- **min** returns the smallest value for the field.

EXAMPLES

The following cursor retrieves all fields from the CITIES record that represents the city of Boston:

```
. sql_135a.epas
       exec sql
         declare legume_village cursor for
           select city, state, altitude, latitude, longitude
           from cities
           where city = ’Boston’;
```

The following cursor retrieves selected fields from CITIES with a population greater than 1,000,000:

```
. sql_135a.epas
       exec sql
         declare big_cities cursor for
           select city, state, population
           from cities
           where population > 1000000;
```

The following cursor joins records from the CITIES and STATES relations:

```
. sql_135a.epas
       exec sql
         declare city_states cursor for
           select c.city, s.state_name
           from states s, cities c
           where s.state = c.state;
```

The following program returns a count of records in the CITIES relation, the maximum population, and the minimum population of cities in that relation:

```
. sql_25c.epas
       program sql (input, output);
```

```
exec sql
  include sqlca;

var  counter    : integer32;
  minpop, maxpop  : integer32;

begin

exec sql
  select count ( * ), max (population), min (population)
    into :counter, :maxpop, :minpop
    from cities;
writeln (’Count: ’, counter);
writeln (’Max Population: ’, maxpop);
writeln (’Min Population: ’, minpop);
end.
```

SEE ALSO

See the entry in this chapter for *predicate*.

NAME

select −selecting records

SYNTAX

select-statement ::= *union-expression* [*ordering-clause*]

union-expression ::= *select-expression* [*into-clause*] [**union** *union-expression*]

ordering-clause ::= **order by** *sort-key-commalist*

sort-key ::= { *database-field* | *integer* } [**asc** | **desc**]

into-clause ::= **into** *host-variable-commalist*

DESCRIPTION

The **select** statement finds the record(s) of the relations specified in the **from** clause that satisfy the given search condition.

You can use the **select** statement by itself or within a **declare cursor** statement:

- Standalone. If the search conditions you specify will return at most one record, you can use the **select** statement by itself. For example, the search condition references a field for which duplicate values have been disallowed.

  returns an error if there is more than one qualifying record.

  Use of the standalone **select** requires the **into** clause.

- Within a **declare cursor** statement. If the search condition identifies an arbitrary number of records, you must define a cursor for retrieval.

  Remember that **declare cursor** is only declarative. Before you can retrieve records via the cursor, you must **open** it and **fetch** records sequentially.

  You cannot use the **into** clause in a **select** statement that appears in a cursor declaration.

ARGUMENTS

union-expression Creates dynamic relations by appending relations. The source relations should have identical structures or at least share some common fields.

ordering-clause Returns the record stream sorted by the values of one or more *database-field*s. You can sort a record stream alphabetically, numerically, by date, or by any combination.

The *database-field* is called the *sort key*. You can construct an *ordering-clause* that includes as many sort keys as you want. Generally speaking, the greater the number of sort keys, the longer it takes for to execute the query.

For each sort key, you can specify whether the sorting order is **asc** (ascending, the default order for the first sort key) or **desc** (descending). The sorting order is ‘‘sticky’’; that is, if you do not specify whether a particular sort key is **asc** or **desc**, assumes that you want the order specified for the last key. Therefore, if you list several sort keys, but only include the word **desc** for the first key, sorts all keys in descending order.

into-clause Specifies the host variables into which you will retrieve database field values. You must preface each host variable with a colon (:). The colon is a convention that indicates the following variable is not a database field.

You cannot use the *into-clause* in a **select** statement that appears inside a cursor declaration.

EXAMPLE

The following **select** statement includes an *ordering-clause* with two sort keys:

```
. sql_137a.epas
        exec sql
          declare urban_population_centers cursor for
            select city, state from cities
            order by state, population desc;
```

The following **select** statement includes an *into-clause* that specifies which database fields are put into which host variables:

```
. sql_137b.epas
        exec sql
          select population, altitude, latitude, longitude
          into :pop, :alt, :lat, :long
          from cities
          where city = ’Boston’;
```

This example assumes that you declared the variables POP, ALT, LAT, and LONG to correspond to the database fields POPULATION, ALTITUDE, LATITUDE, and LONGITUDE from the CITIES relation.

The following cursor declaration joins records from two relations:

```
. sql_138a.epas
        exec sql
          declare city_state cursor for
            select c.city, s.state_name, c.altitude, c.population
            from cities c, states s where c.state = s.state
            order by s.state_name, c.city;
```

The following cursor declaration retrieves the union of two relations:

```
. sql_31c.epas
      exec sql
        declare all_cities cursor for
          select distinct city, state from cities
          union
          select distinct city, state from ski_areas
          union
          select distinct capitol, state from states
          order by 2, 1;
```

The following example retrieves a record from STATES using STATE, a field with unique values:

```
. sql_138c.epas
      exec sql
        select state_name, capitol
          into :statename, :capitol
          from states
          where state = :st;
```

The following example declares a cursor for all items that meet the specified criteria:

```
. sql_138d.epas
      exec sql
        declare middle_america cursor for
          select city, state, population from cities
          where latitude_degrees between 33 and 42
            and longitude_degrees between 79 and 104;
```

SEE ALSO

See the entries in this chapter for:

- *select-expression*
- **open**
- **fetch**
- **close**
- **whenever**

DIAGNOSTICS

The following values may be returned to SQLCODE:

- *SQLCODE < 0* indicates that the statement did not complete.

- *SQLCODE = 0* indicates success.
- *SQLCODE > 0 and < 100* indicates an informational message or warning.
- *SQLCODE = 100* indicates the end of the active set.

See Chapter 6 for a discussion of error handling.

NAME

select-expression −selecting records

SYNTAX

select-clause [*where-clause*] [*grouping-clause*] [*having-clause*]

DESCRIPTION

The *select-expression* specifies the search and delivery conditions for record retrieval.

ARGUMENTS

select-clause Lists the fields to be returned and the source relation or view. The format of the *select-clause* follows:

Syntax: select-clause of Select Expression

select [**distinct**] {*scalar-expression-commalist* | ***** }
from *from-item-commalist*

from-item ::= *relation-name* [*alias*]

An asterisk can be used in place of the full selection list. It is the preferred form for the existential qualifier, **exists**. For example:

```
. no_name
       select city from cities c
         where exists c
           select * from ski_areas
           where city = c.city;
```

However, the wildcard is discouraged for all other uses, because changes to the database (for example, adding or reordering fields), will cause the program to fail after its next precompilation.

The optional keyword **distinct** specifies that only unique values are to be returned. considers the values in the *scalar-expression* list and returns only one set value for each group of records that meets the selection criteria, and that have duplicate values for the *scalar-expression*.

The optional *alias* is used for name recognition, and is associated with a relation. An alias can contain up to 31 characters alphanumeric characters, dollar signs ($), and underscores (_). However, it must start with an alphabetic character. Except for C programs, **gpre** is not sensitive to the case of the alias. For example, it treats **B** and **b** as the same character. For C programs, you can control the case sensitivity of the alias with the **either_case** switch when you preprocess your program.

where-clause Specifies search conditions or combinations of search conditions. The format of the *where-clause* follows:

Syntax: where-clause of Select Expression

where *predicate*

When you specify a search condition or combination of conditions, the condition is evaluated for each record that might qualify . Conceptually, performs a record-by-record search, comparing the value you supplied with the value in the database field you specified. If the two values satisfy the relationship you specified (for example, equals), the search condition evaluates to ‘‘true’’ and that record becomes part of the active set. The search condition can result in a value of ‘‘true,’’ ‘‘false,’’ or ‘‘missing’’ for each record. Such a statement, in which the choice is between the truth or falsity of a proposition, is called a ‘‘Boolean test’’ and is expressed by a *predicate*. See the entry for *predicate* in this chapter.

grouping-clause Partitions the results of the *from-clause* or *where-clause* into control groups, each group containing all rows with identical values for the fields in the *grouping-clause*’s field list. Aggregates in the *select-clause* and *having-clause* are computed over each group. The *select-clause* returns one row for each group.

The aggregate operations are count (**count**), sum (**sum**), average (**avg**), maximum (**max**), and minimum (**min**). See the entry for *scalar-expression* in this chapter.

You can also compute an aggregate value in the *select-clause* and the *having-clause* of the *select-expression*.

Syntax: grouping-clause of Select Expression

group by *database-field-commalist*

The *database-field* specifies the field the values of which you want to group. Each set of values for these fields identifies a group. Chapter 3 discusses the *grouping-clause* in more detail.

having-clause Specifies search conditions for groups of records. If you use the *having-clause*, you must first specify a *grouping-clause*.

Syntax: having-clause of Select Expression

having *predicate*

The *having-clause* eliminates groups of records, while the *where-clause* eliminates individual records. Generally speaking, you can use subqueries to obtain the same results. The main advantage to the use of this clause is brevity. However, some users may find that a more verbose query with subquery is easier to understand.

Chapter 3 discusses the *having-clause* in more detail.

EXAMPLES

The following cursor projects the SKI_AREAS relation on the STATE field:

```
. sql_142a.epas
       exec sql
         declare ski_states cursor for
           select distinct state from ski_areas;
```

The following cursor selects CITIES records for which the POPULATION field is not missing:

```
. sql_142b.epas
       exec sql
         declare inhabited cursor for
           select city, state, population from cities
           where population is not null;
```

The following cursor joins two relations on the STATE field for cities whose population is not missing:

```
. sql_142c.epas
       exec sql
         declare inhabited_join cursor for
           select c.city, s.state_name, c.population
           from cities c, states s
           where c.state = s.state
           and c.population not null;
```

The following cursor calculates the average population by state:

```
. sql_143a.epas
       exec sql
```

```
         declare avg_pop cursor for
         select state, avg (population)
           from cities
           group by state;
```

The following cursor provides a total population by state of municipalities stored in the CITIES relation, but includes only those cities for which the latitude and longitude information has been stored, which are located in states whose names include the word ‘‘New’’, and where the average population of cities in the state exceeds 200,000 people:

```
. sql_143c.epas
       exec sql
         declare total_pop cursor for
           select sum (c.population), s.state_name
           from cities c, states s
           where s.state_name like ’%New%’ and
             c.latitude is not null and
             c.longitude is not null and
             c.state = s.state
           group by s.state
           having avg (population) > 200000;
```

The following program selects the smallest city in each state that has at least two other cities with recorded population. Otherwise, a city would qualify as largest and smallest because it was the only city.

```
. sql_143b.epas
       program sql (input, output);

       exec sql
         include sqlca;

       var
         pop    : integer32;
         city    : array [1..15] of char;
         state_code  : array [1..2] of char;

       begin


       exec sql
         declare small_cities cursor for
         select city, state, population
```

```
        from cities c1
        where c1.population = (
          select min (population)
            from cities c2
            where c2.state = c1.state)
        and 2 <= (
          select count ( * )
            from cities c3
            where c1.state = c3.state
            and c1.city <> c3.city
            and c3.population is not null)
      order by c1.state;

    exec sql
      open small_cities;
    exec sql
      fetch small_cities into :city, :state_code, :pop;

    while sqlcode = 0 do
    begin
      writeln ('The smallest city in ', state_code, ' is ',
        city, ' (pop: ', pop, ')');
      exec sql
        fetch small_cities into :city, :state_code, :pop;
    end;
    exec sql
      close small_cities;
    exec sql
      rollback release;
    end.
```

SEE ALSO

See the entries in this chapter for:

- *predicate*
- *scalar-expression*
- **select**

DIAGNOSTICS

See Chapter 6 for a discussion of error handling.

NAME

update −modify field value

SYNTAX

update *relation-name*
set *assignment-commalist*
[**where** *predicate* | **where current of** *cursor-name*]

assignment ::= *database-field* **=** *scalar-expression*

DESCRIPTION

The **update** statement changes the values of one or more fields in a record in a relation or in the active set of a cursor.

If you do not provide a search condition (**where**...), updates all records in *relation-name*. Be very careful with this option.

ARGUMENTS

relation-name Specifies the relation that contains the record you want to update.

assignment Assigns the *scalar-expression* to *database-field*. This assignment statement belongs to and not to the host language. Do not use a host language assignment or equality operator inside a **update** statement.

If the field you are assigning is a date, you cannot handle the field directly with Instead, you must use date functions such as **gds_$encode_date** and **gds_$decode_date** to convert your external date representation to a host variable in the date format (that is, an array of two 32-bit integers), and then use the assignment to assign the value of the host variable to the database field.

where *predicate* Selects the record to modify.

where current of *cursor-name* Specifies that the current record of the active set is to be modified. If you use the **where current of** clause, updates only the record at which the cursor is pointing. This form of **update** must follow:

- The declaration of the cursor with a **declare cursor** statement
- The opening of that cursor with an **open** statement
- The retrieval of a record from the active set of that cursor with a **fetch** statement

EXAMPLE

The following statement updates the POPULATION field of all records from CITIES that are located in New York:

```
. no_name
       exec sql update cities
         set population = population * 1.03
         where state = ’NY’;
```

The following statement modifies the POPULATION field of all records in the CITIES relation:

```
. no_name
       exec sql update cities
         set population = population * 1.03;
```

The following example declares a cursor, opens it, fetches a record, and then alters that record:

```
. sql_145c.epas
       program popupdate (input_output);
       exec sql
         begin declare section;
       exec sql
         end declare section;

       var
         statecode, st  :   array [1..2] of char;
         cityname  :  array [1..15] of char;
         multiplier  :  integer32;
         pop, new_pop  :  integer32;

       begin

       write (’Enter state with population needing adjustment: ’);
       readln (statecode);
       write (’Percent change (eg 5 => 5% increase; -5 => 5% decrease): ’);
       readln (multiplier);
       multiplier := multiplier + 100;

       exec sql
         declare pop_mod cursor for
           select city, state, population from cities
           where state = :statecode
           for update of population;

       exec sql
         open pop_mod;
       exec sql
```

```
    fetch pop_mod into :cityname, :st, :pop;

  writeln (’ ’);
  while (sqlcode = 0) do
  begin
          new_pop := trunc ((pop * multiplier) / 100);
    writeln (cityname, st, ’ old population: ’, pop,
   ’ new population: ’, new_pop);
    exec sql
      update cities
        set population = :new_pop
        where current of pop_mod;
    exec sql
      fetch pop_mod into :cityname, :st, :pop;
  end;

  exec sql
    close pop_mod;
  exec sql
    rollback release;
  end.
```

SEE ALSO

See the entries in this chapter for:

- *predicate*
- **declare cursor**
- **open**
- **fetch**
- **select**
- **whenever**

DIAGNOSTICS

The following values may be returned to SQLCODE:

- *SQLCODE < 0* indicates that the statement did not complete.
- *SQLCODE = 0* indicates success.
- *SQLCODE > 0 and < 100* indicates an informational message or warning.

See Chapter 6 for a discussion of error handling.

NAME

whenever −handling exceptions

SYNTAX

whenever { **not found** | **sqlerror** | **sqlwarning** }
goto-statement

DESCRIPTION

The **whenever** statement tests the SQLCODE value returned with each execution of an SQL statement. If the listed condition occurs, the **whenever** statement performs the *goto* statement.

The following values may be returned to SQLCODE:

- *SQLCODE < 0* indicates that the statement did not complete. These codes are listed below.
- *SQLCODE = 0* indicates success.
- *SQLCODE > 0 and < 100* indicates an informational message or warning.

A **whenever** statement must precede any statements that might result in an error so that knows what action to take in case of error.

ARGUMENTS

not found Indicates the end of the input stream. This condition corresponds to the SQLCODE value of *100*. This option is useful when you are looping through the active set of a cursor.

sqlerror Indicates that the statement did not complete. This condition corresponds to a negative SQLCODE.

sqlwarning Indicates a general system warning or informational message. This condition corresponds to SQLCODE values between *1* and *99*, inclusive.

EXAMPLE

The following example demonstrates the **sqlerror** option of the **whenever** statement:

```
. sql_131a.epas
       program update_census (input_output);

       label
         error, warn, terminate;

       warn:
         (* since no warnings are defined, fall into error *)
       error:
         writeln ('Encountered SQL error code ', sqlcode);
```

```
    writeln ('Expanded error listing: ');
    gds_$print_status (gds_$status);
    if (sqlcode = -16) then
    begin
      exec sql
        rollback;
      work ();
    end
    else
      exec sql
        rollback release;

  terminate:
  end.
```

DIAGNOSTICS

See Chapter 6 for a discussion of error handling in SQL programs, SQLCODE values, and the corresponding errors.