NAME

based_on −declare program variable

SYNTAX

BASIC syntax:
based_on [*dbhandle*.]*relation-name*.*field-name variable-name*[,*variable-name*]

C syntax:
based_on [*dbhandle*.]*relation-name*.*field-name host-expression*;

host-exp ::= { *host-exp-commalist* | *variable-name* |
**variable-name* | *function* () }

COBOL syntax:
level variable-name **based_on** [*dbhandle*.]*relation-name*.*field-name*

FORTRAN syntax:
based_on [*dbhandle*.]*relation-name*.*field-name variable-name*;

Pascal syntax:
variable **based_on** [*dbhandle*.]*relation-name*.*field-name*

variable ::= { variable-name: | *type-name*: | *function* (*argument-commalist*) }

PL/I syntax:
based_on [*dbhandle*.]*relation-name*.*field-name variable-name*;

DESCRIPTION

The **based_on** clause declares a program variable by referencing a database field. The preprocessor supplies the host variable with all the attributes defined for the database field.

ARGUMENTS

variable Names a host language variable that inherits the characteristics of a database field.

In Pascal, you cannot use the **based_on** clause in a parameter list for a routine. Instead, declare a type and then declare the formal parameter to be that type.

dbhandle Specifies the source of the database field. The database handle must have been declared in an earlier **database** statement.

relation-name.*field-name* Specifies the relation and field on which to base the host variable.

EXAMPLE

The following example shows two **based_on** declarations as they would appear in a C program:

```
based_on states.state_name state_name;
        based_on states.capitol capitol_city;
```

The following example shows the **based_on** declaration as it would appear in a Pascal program:

```
var state : based_on states.state;
```

SEE ALSO

Host language documentation for declaration of variables.

DIAGNOSTICS

See Chapter 8 for a discussion of errors and error handling.

NAME

boolean-expression −relationship between value expressions

SYNTAX

boolean-expression ::= { [**not**] *conditional-expression* |
conditional-expression **and** *conditional-expression* |
conditional-expression **or** *conditional-expression* }

conditional-expression ::= { *comparison-condition* | *between-condition* |
starting-condition | *containing-condition* |
matching-condition | *not-condition* | *unique-condition* }

DESCRIPTION

A *boolean-expression* evaluates to true, false, or missing. It describes the characteristics of a single value expression (for example, a missing value) or the relationship between two value expressions (for example, *x* is greater than *y*).

The order of precedence for evaluating compound Boolean expressions is **not**, **and**, and **or**.

ARGUMENTS

comparison-condition Describes the characteristics of a single expression. The format of the *comparison-condition* follows:

Syntax: comparison-condition of Boolean Expression

value-expression-1 relational-operator value-expression-2

The *relational-operator* can be any of the operators in the following table:

Operator	Relationship
eq *or* = *or* ==	equal
ne *or* <> *or* !=	not equal
gt *or* >	greater than
ge *or* >=	greater than or equal
lt *or* <	less than
le *or* <=	less than or equal

between-condition Tests whether a value expression, *value-expression-1*, occurs between two other value expressions, *value-expression-2* and *value-expression-3*. This test is inclusive of the boundary values. The format of the *between-condition* follows:

Syntax: between-condition of Boolean Expression

value-expression-1 [**not**] **between**
value-expression-2 **and** *value-expression-3*

containing-condition Tests for the presence of *string* (case-*in*sensitive) anywhere in *value-expression*. It evaluates to true if *string* is contained in *value-expression*. If the value of *value-expression* is missing, the result is missing. The format of the *containing-condition* follows:

Syntax: containing-condition of Boolean Expression

value-expression-1 [**not**] **containing** *value-expression-2*

starting-condition Tests for the presence of *string* (case-sensitive) at the beginning of *value-expression*. It evaluates to true if the first characters of *value-expression* match *string*. The search is case-sensitive. The

format of the *starting-condition* follows:

Syntax: starting-condition Boolean Expression

value-expression-1 [**not**] **starting with** *value-expression-2*

matching-condition Tests for the presence of *wildcarded-string*, a string that can contain the wildcard characters ***** and **?**. The asterisk matches an unspecified run of characters, while the question mark matches a single character. This test is case *in*sensitive. The format of the *matching-condition* follows:

Syntax: matching-condition of Boolean Expression

value-expression-1 [**not**] **matching** *value-expression-2*

missing-condition Tests for the absence of a value in *dbfield-expression*. It is true if the value of *dbfield-expression* is missing. The format of the *missing-condition* follows:

Syntax: missing-condition of Boolean Expression

dbfield-expression [**not**] **missing**

Unless you specify otherwise in the field's definition, blanks are returned for numbers, characters, and dates, and nothing is returned for blobs. See for more information about defining alternate missing values.

any-condition Tests for the existence of at least one qualifying record in a relation or relations. This expression is true if the record stream specified by *rse* includes at least one record. If you add **not**, the expression is true if there are *no* records in the record stream. The format of the *any-condition* follows:

Syntax: any-condition of Boolean Expression

[**not**] **any** *rse*

You might want to use **any** instead of joining records if all you want to do is establish that a record exists. As soon finds one record that meets the search criteria, it stops, whereas a join would continue until it found all qualifying records.

unique-condition Tests for the existence of exactly one qualifying record. This expression is true if the record stream specified by *rse* consists of only one record. If you add **not**, the condition is true if there is more than one record in the record stream or if the record stream is empty. The format of the *unique-condition* follows:

Syntax: unique-condition of Boolean Expression

[**not**] **unique** *rse*

EXAMPLES

The following statement looks for cities with populations between 100,000 and 250,000:

```
for c in cities with c.population between 100000 and 250000
          writeln (c.city, c.state, c.population);
        end_for;
```

The following statement looks for cities with the substring ‘‘ville’’ somewhere in their name:

```
for c in cities with c.city containing 'ville'
          writeln (c.city, c.state);
        end_for;
```

The following statement looks for cities that start with the string *New*:

```
for c in cities with c.city starting with 'New'
          writeln (c.city, c.state);
        end_for;
```

The following statement looks for cities with the string ‘‘ton’’ following any number of other characters:

```
for c in cities with c.city matching '*ton*'
          writeln (c.city, c.state);
        end_for;
```

The following statement looks for states with the state abbreviation equal to ‘‘N’’ followed by exactly one character:

```
for c in cities with c.city matching 'N?'
          writeln (c.city, c.state);
        end_for;
```

The following statement looks for states that have a missing value for the CAPITOL field:

```
for s in states with s.capitol missing
          writeln (s.state_name);
        end_for;
```

The following statement prints the name of any state for which there are cities stored:

```
. gdml_120a.epas
       for s in states with any c in cities with
         c.state = s.state
         writeln (s.state_name);
       end_for;
```

The following query prints the names of states that have only one ski area:

```
. gdml_120b.epas
       for s in states with unique ski in ski_areas with
         ski.state = s.state
         writeln (s.state_name);
       end_for;
```

SEE ALSO

See the entries in this chapter for:

- *value-expression*
- *rse* in this chapter.

DIAGNOSTICS

See Chapter 8 for a discussion of errors and error handling.

NAME

close_blob −close blob field

SYNTAX

close_blob *blob-variable* [*on-error*]

on-error ::= **on_error** *statement...* **end_error**

DESCRIPTION

The **close_blob** statement closes an open blob field and releases system resources associated with blob retrieval or update.

You should close the blob as soon as you finish reading or writing. If you fail to close a blob to which you wrote data, you will not be able to make the blob permanent. Closing a blob is especially important when you access remote databases. Because remote interface buffers segment transfer between participating nodes, it may truncate the last segment you write unless you explicitly signal that the blob is closed.

Once you close a blob, you cannot read from or write to that blob without re-opening it with an **open_blob** or **for blob** statement.

ARGUMENTS

blob-variable A temporary name used for name recognition. It is associated with individual segments in the field and is used very much like a context variable. You must have assigned the blob variable in an earlier **create_blob** or **open_blob** statement.

on-error Specifies the action to be performed if an error occurs during the close operation.

EXAMPLE

The following program creates a record stream from two relations, opens a blob field, reads segments from the blob field, and then closes the blob field:

```
. gdml_119a.epas
        program update_guide (input, output);

        database atlas = filename 'atlas.gdb';

        begin

        ready atlas;
        start_transaction;
        for s in states cross t in tourism over state sorted by s.state
        begin
          writeln (s.state_name, ' ', t.city);
```

```
  open_blob b in t.office
  get_segment b;
  while (gds_$status [2] = 0) or (gds_$status [2] = gds_$segment) do
  begin
    write (b.segment:b.length);
    get_segment b;
  end;
  writeln;
  close_blob b;
end;
end_for;
commit;
finish atlas;
end.
```

SEE ALSO

For some guidance on the best approach to processing blobs, see Chapter 6. See also the entries in this chapter for:

- **open_blob**
- **on_error**

DIAGNOSTICS

See Chapter 8 for a discussion of errors and error handling.

NAME

commit −write changes to database

SYNTAX

commit [*transaction-handle*] [*on-error*]

on-error ::= **on_error** *statement...* **end_error**

DESCRIPTION

The **commit** statement ends a transaction and makes the transaction's changes visible to other users.

The **commit** statement affects all databases in the transaction, writing to the database(s) all changes made during the transaction. It flushes all modified buffers and closes any record streams that are open.

ARGUMENTS

transaction-handle Specifies the transaction you want to commit. If the transaction you want to commit has a transaction handle associated with it, you must use that handle when you commit the transaction.

If you do not specify a transaction handle on a **commit** statement, commits the ''default'' transaction. The default transaction is what starts when you use a **start_transaction** statement without a handle.

on-error Specifies the action to be performed if an error occurs during the commit operation.

EXAMPLE

The following Pascal example starts an unnamed transaction, performs some unspecified data manipulation, and then writes the changes to the database:

```
start_transaction concurrency;
          .
          .
          .
        commit;
```

The following Pascal program starts two separate transactions, one to get a badge number, and the other to store a new employee. This simplified program contains no error handling.

```
. gdml_122a.epas
        program map (input_output);

        database db = filename 'emp.gdb';

        type  badge_type  = based on badge_num.badge;
```

```
var
  store_emp_tr  : gds_$handle  := nil;
        to_be_stored  : integer;

function get_badge   : badge_type;
var
  get_badge_tr  : gds_$handle;
begin
        get_badge_tr := nil;
  start_transaction get_badge_tr;
  for (transaction_handle get_badge_tr) b in badge_num
    get_badge := b.badge;
    modify b using
      b.badge := b.badge + 1;
    end_modify;
  end_for;
  commit get_badge_tr;
end;    { function get_badge }

begin
        ready;
  start_transaction store_emp_tr;
   write (’Enter the number of new employees: ’ );
  readln (to_be_stored);
  while to_be_stored > 0 do
  begin
    store (transaction_handle store_emp_tr) e in employees using
      e.badge := get_badge;
      write (’Enter first name: ’);
      readln (e.first_name);
      write (’Enter last name: ’);
      readln (e.last_name);
      write (’Enter supervisor”s id: ’);
      readln (e.supervisor);
      write (’Enter department: ’);
      readln (e.department);
    end_store;
                to_be_stored := to_be_stored - 1;
   end;
  commit store_emp_tr;
  finish;
end.
```

SEE ALSO

See the entries in this chapter for:

- **start_transaction**
- *transaction-handle*
- **on_error**

DIAGNOSTICS

See Chapter 8 for a discussion of errors and error handling.

NAME

create_blob −create blob

SYNTAX

create_blob *blob-variable* **in** *dbfield-expression* [*on-error*]

on-error ::= **on_error** *statement...* **end_error**

DESCRIPTION

The **create_blob** statement creates a blob.

ARGUMENTS

blob-variable A temporary name used for name recognition. It is associated with individual segments in the field and is used much like a context variable.

dbfield-expression A value expression that identifies a field containing blob data.

on-error Specifies the action to be performed if an error occurs during the creation of the blob field.

EXAMPLE

The following example creates a record stream, creates a blob field, and writes segments to the blob field:

```
. blob_6.epas
        program store_tour (input_output);
        database db = filename 'atlas.gdb';
        var  i : integer;
        var  statecode based_on states.state
        .sp
        begin
        store t in tourism using
          write (statecode)
          t.state :=
          t.date_modified.char [6] := 'TODAY';
          create_blob b in t.blurb;
          writeln ('Enter new blurb one line at a time');
          writeln (' A line containing "-30-" ends input');
          readln (b.segment);
          while (b.segment <> '-30-') do
            begin
              for i := sizeof (b.segment) downto 1 do
                if b.segment[i] <> ' ' then exit;
              i := i + 1;
```

```
      b.segment[i] := chr(10);
      b.length := i;
      put_segment b;
      readln (b.segment);
    end;
  close_blob b;
end_store;

rollback;
finish;

end.
```

SEE ALSO

For some guidance on the best approach to processing blobs, see Chapter 6 in this manual. See also the entries in this chapter for:

- **on_error**
- *value-expression*

DIAGNOSTICS

See Chapter 8 for a discussion of errors and error handling.

NAME

database −declare database

SYNTAX

database *database-handle* = [*declaration-scope*]
[**compiletime**] [**filename**] *database-filespec*

[**runtime** [**filename**] {*database-filespec* | *host-variable*}]

declaration-scope ::= { **static** | **extern** }

DESCRIPTION

The **database** declaration specifies the database to be accessed by a program or program module. Because the **database** declaration identifies the source of metadata, it must precede any database access. However, it is the **ready** statement or equivalent action that actually opens the database for access.

The **database** declaration optionally supports the specification of a *runtime* database. However, the runtime database is more appropriately referenced in (and opened by) the **ready** statement. For example, your program may access a number of databases that use common metadata, but contain different data. Applications of this type include CAD/CAM and test control systems, in which a boilerplate database supplies metadata (for example, relations for wing struts and other aircraft assemblies) while instances of that database contain actual data (individual databases for IL-62, IL-70, and IL-82 aircraft designs). The use of the boilerplate database for metadata helps ensure that you are keeping track of the same data for all your aircraft designs.

However, if you find that the runtime database is always the same, and different from the compiletime database, you can add the **runtime** clause to the **database** declaration. If you choose only one **compiletime** identifier, **gpre** uses that identifier for both compilation and runtime unless you provide a runtime file in the **ready** statement.

ARGUMENTS

database-handle Declares a name that you can use when you have to reference multiple databases in a program.

N O T E

Many of the examples in this manual use the database handle DB, *which is a reserved word in VAX COBOL.* You cannot use reserved words as database handles.

declaration-scope Declares the scope of the handle specified by the *database-handle* clause. If you do not specify a declaration scope, the scope of the handle defaults to **global**.

If you specify **static**, the scope is the module containing the **database** declaration.

If you specify **extern**, the handle will correspond to one in another module with a **global** scope.

If all database handles in a module have the same scope, the handle for the default transaction will also have that scope; otherwise, the handle for the default transaction will have a **global** scope.

database-filespec Specifies the database from which the preprocessor reads the metadata. The *database-filespec* can be:

- A filename enclosed in single (') or double (") quotation marks, depending on your host language conventions.
- A logical name that resolves to a quoted file specification.

The file specification can contain the full pathname, including the name of the node on which the database is stored. If you are in a directory other than the one that contains the database file, the file specification *must* include the pathname. If the database is on another node, the *filespec* must include the node name and pathname. You can define a link or logical name for the database file.

File specifications for remote databases have the following form:

Syntax: Remote Database File Specification

VMS to ULTRIX:
node-name::*filespec*

VMS to non-VMS and non-ULTRIX:
node-name^filespec

Within Apollo DOMAIN:
//*node-name*/*filespec*

All Else:
node-name:*filespec*

Make sure that what follows the colon is a valid file specification on the target system; use brackets, slashes, and spaces as appropriate.

EXAMPLE

The following Pascal program includes two **database** declarations:

```
program mapper (input, output);
        database atlas =
```

```
  compiletime filename 'atlas.gdb';
database gazetteer =
  compiletime filename '/usr/gds/examples/atlas.gdb';
begin

ready atlas;
ready gazetteer;

for s in atlas.states sorted by s.state
  begin
    writeln (s.state);
    for c in gazetteer.cities with c.state = s.state
      writeln (c.city, c.latitude, c.longitude);
    end_for;
  end;
end_for;
end.
```

SEE ALSO

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

DIAGNOSTICS

See Chapter 8 for a discussion of errors and error handling.

NAME

erase −delete record

SYNTAX

erase *context-variable* [*on-error*]

on-error ::= **on_error** *statement...* **end_error**

DESCRIPTION

The **erase** statement removes records from an open record stream.

You cannot erase records from views or joins. Rather, you must erase them through the source relations.

ARGUMENTS

context-variable Specifies the record stream from which to erase the record(s). You must declare the *context-variable* in a **for** or **start_stream** statement.

on-error Specifies the action to be performed if an error occurs during the erase operation.

EXAMPLE

The following statements prompt for a field value and then delete records with that value:

```
var statecode: based_on states.state;
          .
          .
          .
        write ('State to depopulate: ');
        readln (statecode);

        for c in cities with c.state = statecode
          erase c;
        end_for;
```

SEE ALSO

See the entries in this chapter for:

- **on_error**
- **for**
- **start_stream**

DIAGNOSTICS

See Chapter 8 for a discussion of errors and error handling.

NAME

fetch −advance record stream pointer

SYNTAX

fetch *stream-name* [**at end** *statement...* **end_fetch**] [*on-error*]

on-error ::= **on_error** *statement...* **end_error**

DESCRIPTION

The **fetch** statement advances the record stream pointer to the next record in a record stream, thus selecting the current record of that stream for whatever retrieval or manipulation operation you choose.

The **fetch** statement:

- Can be used only in a record stream created by a **start_stream** statement.
- Must precede any other statement that affects the current record.

ARGUMENTS

stream-name Specifies the stream from which to fetch records. You must open the stream with a **start_stream** statement.

statement Specifies or host language statements to be executed on each record in the stream.

on-error Specifies the action to be performed if an error occurs during the fetch operation.

at end Specifies the action to be taken when the program reaches the end of the stream. If you include more than one *statement*, you must separate them using the host language convention.

EXAMPLE

The following program demonstrates the use of the **start_stream** statement in a loop that may be terminated by user interaction:

```
. gdml_130a.epas
        program map (input_output);

        database db = filename 'atlas.gdb';
        var  end_of_stream  : boolean;
          genug    : char;

        begin
          start_stream geodata using c in cities
            sorted by c.latitude, c.longitude;
```

```
  end_of_stream := false;
  fetch geodata
    at end end_of_stream := true;
  end_fetch;
  while not end_of_stream do begin
    writeln (c.latitude, c.longitude, c.altitude,
      c.city, c.state);
    write ('Seen enough? (Y/N) ');
    readln (genug);
    if genug = 'Y' then
      end_of_stream := true;
    fetch geodata
      at end begin
        end_of_stream := true;
        writeln ('Sorry, there is no more.');
      end;
    end_fetch;
  end;
  end_stream geodata;
  commit;
  finish;
end.
```

SEE ALSO

See the entries in this chapter for:

- **start_stream**
- **on_error**

DIAGNOSTICS

See Chapter 8 for a discussion of errors and error handling.

NAME

finish −close database

SYNTAX

finish [*database-handle-commalist*] [*on-error*]

on-error ::= **on_error** *statement...* **end_error**

DESCRIPTION

The **finish** statement closes either the default database (that is, a database opened without a database handle) or a specific database identified by a database handle.

ARGUMENTS

database-handle Specifies which open database or databases you want to close. A **database** declaration declares this handle.

- If you use the optional *database-handle* clause, the database handle must have been previously associated with a database in the **database** declaration. This clause lets you close specific databases if you are using multiple databases in your program.
- If you do *not* specify a database handle, the **finish** statement commits the default transaction. If you want to close a specific database, you must first commit or roll back the transaction.
- Non-default transactions that have not been committed are automatically rolled back by a **finish** statement.

on-error Specifies the action to be performed if an error occurs during the finish operation.

EXAMPLE

The following statement closes any open databases:

```
finish;
```

The following statements close the databases identified by the handle:

```
finish atlas;
        finish mapper;
        finish your_broccoli;
```

SEE ALSO

See the entries in this chapter for:

- **on_error**

- **database**
- **commit**
- **rollback**

DIAGNOSTICS

See Chapter 8 for a discussion of errors and error handling.

NAME

for −loop structure

SYNTAX

for [*request-option*] *rse*
statement...
end_for [*on-error*]

on-error ::= **on_error** *statement...* **end_error**

DESCRIPTION

The **for** statement executes a statement or group of statements once for each record in a stream formed by a record selection expression.

You can nest **for** loops to display a hierarchy of records or to join relations across databases.

ARGUMENTS

request-option Specifies a transaction handle and/or request handle that determine the transaction and/or request in which the **for** loop executes.

rse Specifies the record selection criteria used to create the record stream.

The scope of a context variable declared in the *rse* is the statement in which it was declared. Therefore, you can re-use a context variable from a **for** statement when you end the for loop with **end_for** and begin a new record stream with a **for** or **start_stream** statement.

You cannot reference more than one database in a record selection expression. Therefore, use nested **for** loops to join relations across databases.

on-error Specifies the action to be performed if an error occurs during the **for** loop.

statement Specifies GDML or host language statements to be executed within the **for** loop. The *statement*s you include in a **for** loop are subject to the following rules:

- You can nest **for** statements within other **for** statements.
- If you include more than one *statement*, you must separate them using the host language convention.
- If you use other GDML statements in the for loop, those statements can use the context variables declared in the **for** statement or in an outer statement, as well as contexts declared in the current **for** statement.

EXAMPLE

The following statements retrieve records through a **for** loop:

```
for c in cities with population gt 1000000
          writeln (c.city, c.state, c.population);
        end_for;
```

The following statements join two relations using a **for** loop:

```
for c in cities cross s in states with c.state = s.state
          writeln (c.city, s.state_name, c.population);
        end_for;
```

The following statements use an outer **for** loop to create a record stream from which a **store** statement takes some values, host variables supply some values, and unreferenced fields are set to missing:

```
for oldcity in cities with oldcity.city = hostvar1
          store newcity in cities using
            newcity.city = hostvar2;
            newcity.state = oldcity.state;
            newcity.population = oldcity.population * hostvar3;
            newcity.altitude = oldcity.altitude;
          end_store;
        end_for;
```

The following example lists employees by department:

```
. gdml_136a.epas
        program print_depts (input, output);
        database db = filename 'emp.gdb';

        begin
          for d in departments sorted by d.dept_name
          begin
            writeln (d.department, ' manager: ', d.manager);
            for e in employees with e.department = d.department
              sorted by e.badge
              writeln ('      ', last_name, first_name);
            end_for;
          end_for;
        end.
```

The next example demonstrates the way to join relations across databases. It uses two copies of the sample atlas databases, one of which is in your current directory and the other in the examples directory provided with The statements display values from the STATES relation in one copy of the atlas database, and values stored in another database from CITIES in those states. The join term is the STATE field in both relations.

```
program mapper (input, output);
        database atlas =
          compiletime filename 'atlas.gdb';
        database gazetteer =
          compiletime filename '/usr/gds/examples/atlas.gdb';
        begin

        ready atlas;
        ready gazetteer;

        for s in atlas.states sorted by s.state
          begin
            writeln (s.state);
            for c in gazetteer.cities with c.state = s.state
              writeln (c.city, c.latitude, c.longitude);
            end_for;
          end;
        end_for;

        finish atlas;
        finish gazetteer;

        end.
```

The following program hires everybody's offspring and assigns them new badge numbers. Note that each request (that is, each **for** and **store**) must use the same request options, even though they are nested. The **modify** statement is not a separate request and does not require a transaction handle. The outer **for** statement is in the default transaction so that it will not read the newly stored records and start prompting for employee grandchildren.

```
. gdml_137a.epas
        program nested_for (input, output);
        database db = filename 'emp.gdb';

        var
          update_tr  : gds_$handle := nil;
          check    : char;
          fnl, lnl  : integer;
```

```
begin
ready;
start_transaction update_tr consistency read_write reserving
  badge_num, employees for protected write;

start_transaction;

for e in employees
  fnl := 1;
  while (e.first_name [fnl] <>  ' ') do
    fnl := fnl + 1;
  lnl := 1;
  while (e.last_name [lnl] <>  ' ') do
    lnl := lnl + 1;
  write ('Should we hire ', e.first_name:fnl,
    e.last_name:lnl-1, '"s kid? ');
  readln (check);
  if (check = 'y') or (check = 'Y') then
  begin
    for (transaction_handle update_tr) b in badge_num
    begin
      store (transaction_handle update_tr) n_e in employees using
      begin
        write ('What"s the kid"s first name? ');
        readln (n_e.first_name);
        n_e.last_name := e.last_name;
        write ('What"s the kid"s date of birth? ');
        readln (n_e.birth_date.char[20]);
        n_e.badge := b.badge + 1;
        n_e.department := 'NEP';
        n_e.supervisor := 13;
      end;
      end_store;
      modify b using
        b.badge := b.badge + 1;
      end_modify;
    end;
    end_for;
  end;
end_for;
commit update_tr;
commit;
finish;
```

```
end.
```

SEE ALSO

See the entries in this chapter for:

- *request-option*
- *rse*
- **on_error**
- **for blob**

DIAGNOSTICS

See Chapter 8 for a discussion of errors and error handling.

NAME

for blob −access blob field

SYNTAX

for *blob-variable* **in** *dbfield-expression* [*on-error*]
statement...
end_for

on-error ::= **on_error** *statement...* **end_error**

DESCRIPTION

The **for blob** statement retrieves data from a field that contains blob data.

The **for blob** statement is the easiest way to access blobs. You should use it when you process whole segments of a blob field or the entire contents of the blob buffer, without calling special formatting routines.

To read or write a blob field with the **for blob** statement:

- Construct a loop with the ‘‘other’’ **for** statement. This *outer for loop* creates a record stream.
- Construct a loop with the **for blob** statement. This *inner loop* swings through the blob, returning a segment at a time.
- Perform whatever action(s) you want to the blob under the control of the inner loop.
- Return control to the outer loop when you are finished with the blob field.

ARGUMENTS

blob-variable A temporary name used for name recognition. It is associated with individual segments in the field and is used very much like a context variable.

dbfield-expression A value expression that identifies a field containing blob data.

on-error Specifies the action to be performed if an error occurs during the **for** loop.

statement Any valid host language or statement. Use host language punctuation to terminate each statement.

EXAMPLE

The following statements create a record stream, display several structured fields from those records, and display a blob from each of those records:

```
for tour in tourism sorted by tour.state
         writeln (tour.city, tour.state, tour.zip);
         writeln;
```

```
  for blob in tour.guidebook
    write (blob.segment:blob.length);
  end_for; {blob loop}
       writeln;
end_for; {for loop}
```

The following program copies a blob to another database by retrieving it in a **for blob** statement:

```
. gdml_139a.epas
  program update_guide (input, output);

  database atlas = filename 'atlas.gdb';
  database guide = filename 'coastal_guide.gdb';

  begin
  start_transaction;

  for t in atlas.tourism
  begin
    store new in guide.tourism using
      new.state := t.state;
      new.city := t.city;
              new.zip := t.zip;
      create_blob n_guide in new.guidebook;
      for o_guide in t.guidebook
        n_guide.segment := o_guide.segment;
        n_guide.length := o_guide.length;
        put_segment n_guide;
      end_for;
      close_blob n_guide;
    end_store;
  end;
  end_for;
  commit;
  finish;
  end.
```

SEE ALSO

For more guidance on processing blobs, see Chapter 6 of this manual. See also the entries in this chapter for:

- **on_error**
- *value-expression*

DIAGNOSTICS

See Chapter 8 for a discussion of errors and error handling.

NAME

get_segment −retrieve blob segment

SYNTAX

get_segment *blob-variable* [*on-error*]

on-error ::= **on_error** *statement...* **end_error**

DESCRIPTION

The **get_segment** statement reads a portion of a blob field. Before you can read a blob, you must open it with an **open_blob** statement.

ARGUMENTS

blob-variable A temporary name used for name recognition. It is associated with individual segments in the field and is used like a context variable. You must have assigned the blob variable in an earlier **open_blob** statement.

on-error Specifies the action to be performed if an error occurs during the get operation.

EXAMPLE

The following example creates a record stream, opens a blob field, and reads segments from the blob field:

```
for tour in tourism cross s in states over state
            sorted by s.state
          writeln (tour.zip, s.state_name, s.area);
          open_blob b in tour.guidebook;
          get_segment b;
          while (gds_$status [2] = 0) or
            (gds_$status [2] = gds_$segment) DO
          begin
            write (b.segment : b.length);
            get_segment b;
          end;
          close_blob b;
          writeln;
        end_for;
```

SEE ALSO

For some guidance on the best approach to processing blobs, see Chapter 6 in this manual. See also the entries in this chapter for:

- **open_blob**

- **close_blob**
- **on_error**

DIAGNOSTICS

See Chapter 8 for a discussion of errors and error handling.

NAME

modify −change field value

SYNTAX

modify *context-variable* **using**
statement...
end_modify [*on-error*]

on-error ::= **on_error** *statement...* **end_error**

DESCRIPTION

The **modify** statement updates a field or fields in a record from a record stream.

You cannot modify records through views. Rather, you must modify them through the source relations.

If the field you want to modify contains blob data, use the **put_segment** statement to modify it.

ARGUMENTS

context-variable Specifies the record stream from which the record is to be modified. You must declare *context-variable* in a **for** or **start_stream** statement.

on-error Specifies the action to be performed if an error occurs during the modify operation.

statement Specifies the action to be taken in modifying the record(s). The *statement*s are typically assignments. If you include more than one *statement*, you must separate them using the host language convention.

EXAMPLE

The following statements increase the value of the POPULATION field in all cities in a given state:

```
var statecode : based_on states.state;
        write ('State code [2 characters, uppercase]: ');
        readln (statecode);

        for c in cities with c.state = statecode
          modify c using
            c.population := c.population * 1.2;
          end_modify;
        end_for;
```

SEE ALSO

See the entries in this chapter for:

- **on_error**
- **for**
- **start_stream**
- **put_segment**

DIAGNOSTICS

See Chapter 8 for a discussion of errors and error handling.

NAME

on_error −error handling

SYNTAX

on_error
statement...
end_error

DESCRIPTION

The **on_error** clause specifies the action the program will take if an error occurs during the execution of the associated operation.

All statements can include an **on_error** clause; the **database** declaration cannot.

ARGUMENTS

statement A host language or statement. If you include more than one *statement*, you must separate them using the host language convention.

EXAMPLE

The following program changes the type of ski areas, using reasonable error handling. It prompts for the name of a database and reprompts if there is an error during the **ready** statement. The modification takes place in a subroutine that returns the status of the change. Validation errors are handled in the routine, thus avoiding restarting either the transaction or the **for** loop. Deadlocks are handled by the main routine, which rolls back and retries. Other errors print the status, rollback and exit.

```
. gdml_145a.epas
        program ski_areas (input_output);

        database db = filename 'atlas.gdb';

        type
          name    = based on ski_areas.name;
          a_type    = based on ski_areas.type;
        var
                more    : char := 'y';
          area_name   : name;
          area_type  : a_type;
          stat    : integer;

        function modify_type (area_name : name; area_type : a_type) : integer;
        label
          re_mod;
```

```
begin
  modify_type := gds_$true;
  start_transaction;
  for ski in ski_areas with ski.name = area_name
re_mod:
    modify ski using
      ski.type := area_type;
    end_modify
    on_error
    begin
      if gds_$status [2] = gds_$not_valid then
      begin
        writeln ('Type must be N, A, or B');
        write ('Enter new area type: ');
        readln (area_type);
        goto re_mod;
      end
      else if gds_$status [2] <> gds_$deadlock then
        gds_$print_status (gds_$status);
      modify_type := gds_$false;
       rollback;
      return;
    end;
     end_error;
  end_for
  on_error
    if gds_$status [2] <> gds_$deadlock then
      gds_$print_status (gds_$status);
    modify_type := gds_$false;
  end_error;
  commit;
end;

function open_database   : integer;
var
  filename  : array [1..40] of char;

begin
  open_database := 0;
  write ('Please enter pathname of database ("quit" to exit): ');
  readln (filename);
  if filename = 'quit' then
    open_database := -1
```

```
  else begin
    ready filename as db
    on_error
    begin
      writeln ('Error during database open.  Status follows.');
      gds_$print_status (gds_$status);
      writeln;
      open_database := 1;
    end;
    end_error;
  end;
end;

begin
  repeat
  begin
     stat := open_database;
    if stat = -1 then
    begin
      writeln ('Toodles, kid!');
      return;
    end;
  end;
  until (stat = 0);
  while more = 'y' do
  begin
    write ('Enter ski_area name: ');
    readln (area_name);
    write ('Enter new area type: ');
    readln (area_type);
    stat := modify_type (area_name, area_type);
    while stat = gds_$false do
    begin
      if gds_$status [2] = gds_$deadlock then
        stat := modify_type (area_name, area_type)
      else
      begin
        writeln ('Farewell, cruel world...');
        finish;
        return;
      end;
    end;
    write ('Enter "y" to change another record: ');
```

```
    readln (more);
  end;
  finish;
end.
```

DIAGNOSTICS

See Chapter 8 for a discussion of errors and error handling.

NAME

open_blob −open blob field for access

SYNTAX

open_blob *blob-variable* **in** *dbfield-expression* [*on-error*]

on-error ::= **on_error** *statement...* **end_error**

DESCRIPTION

The **open_blob** statement opens a blob so that its data may be retrieved.

You can process blobs by using the following statements:

- **create_blob**
- **for blob**
- **open_blob**
- **get_segment** and **put_segment**
- **close_blob**

This approach is most useful when you process a blob field, look at what is in the data, and make decisions based on the contents.

ARGUMENTS

blob-variable Declares a temporary name to be used for name recognition. It is associated with individual segments in the field and is used like a context variable.

dbfield-expression A value expression that identifies a field containing blob data. The context variable must be assigned in an outer **for** loop or **start_stream** statement.

on-error Specifies the action to be performed if an error occurs during the blob operation.

EXAMPLE

The following example creates a record stream from two relations, opens a blob field, and reads segments from the blob field:

```
for tour in tourism cross s in states over state
            sorted by s.state
          writeln (tour.zip, s.state_name, s.area);
          open_blob b in tour.guidebook;
          get_segment b;
```

```
  while (gds_$status [2] = 0) DO
  begin
    write (b.segment : b.length);
    get_segment b;
  end;
  close_blob b;
  writeln;
end_for;
```

SEE ALSO

For more guidance on processing blobs, see Chapter 6 of this manual. See also the entries in this chapter for:

- **on_error**
- **for blob**
- **close_blob**

DIAGNOSTICS

See Chapter 8 for a discussion of errors and error handling.

NAME

prepare −prepare to commit transaction

SYNTAX

prepare [*transaction-handle*] [*on-error*]

on-error ::= **on_error** *statement...* **end_error**

DESCRIPTION

The **prepare** statement signals your intention to commit either the default transaction (that is, a transaction you start without declaring a handle) or the transaction specified by the optional transaction handle.

The **prepare** statement executes the first phase of a two-phase commit. The access method polls all participants and waits for replies from each. It checks to see that no other database activity can affect the transaction. The **prepare** statement is particularly useful for transactions that access multiple databases or for transactions that involve both database and non-database activity.

If the statement completes successfully, guarantees that a **commit** statement will execute successfully if the disk is still intact.

ARGUMENTS

transaction-handle Specifies which transaction to prepare to commit. If the transaction you want to commit has a transaction handle associated with it, you must use that handle on the **prepare** and subsequent **commit** statements.

If you do not specify a handle on the **prepare** statement, prepares the ‘‘default’’ transaction. The default transaction is what gets started when you use a **start_transaction** statement without a handle.

on-error Specifies the action to be performed if an error occurs during the prepare operation.

EXAMPLE

The following extract includes a **prepare** statement with an **on_error** clause:

```
prepare zip_code_update
          on_error
          begin
            writeln (’Something failed during prepare’);
            gds_$print_status (gds_$status);
            writeln (’Starting rollback...’);
            rollback zip_code_update;
            goto failure;
                end
          end_error;
```

```
commit zip_code_update;
```

SEE ALSO

See the entries in this chapter for:

- **commit**
- **on_error**
- *transaction-handle*

DIAGNOSTICS

See Chapter 8 for a discussion of errors and error handling.

NAME

put_segment −write a blob segment

SYNTAX

put_segment *blob-variable* [*on-error*]

on-error ::= **on_error** *statement...* **end_error**

DESCRIPTION

The **put_segment** statement writes a portion of a blob field.

Before you can write a blob field, you must create it with a **create_blob** statement.

ARGUMENTS

blob-variable A temporary name used for name recognition. It is associated with individual segments in the field and is used much like a context variable. You must have assigned the blob variable in an earlier **open_blob** statement.

on-error Specifies the action to be performed if an error occurs during the put operation.

EXAMPLE

The following statements create a record stream, create a blob field, and write segments to the blob field:

```
. {blob_3.epas}
        store tour in tourism using
          write (’Enter state code: ’);
          readln (tour.state)
          write (’Enter zip code: ’);
          readln (tour.zip)
          write (’Enter city: ’);
          readln (tour.city)
          create_blob b in tour.guidebook;
          writeln (’Enter new blurb one line at a time’);
          writeln (’ A line containing ”-30-” ends input’);
          readln (b.segment);
          while (b.segment <> ’-30-’) do
            begin
              for i := sizeof (b.segment) downto 1 do
                if b.segment[i] <> ’ ’ then exit;
              i := i + 1;
              b.segment[i] := chr(10);
              b.length := i;
```

```
      put_segment b;
      readln (b.segment);
    end;
  close_blob b;
end_store;
```

The following program copies the contents of a blob field from one database to another:

```
. gdml_139a.epas
 program update_guide (input, output);

database atlas = filename 'atlas.gdb';
database guide = filename 'coastal_guide.gdb';

begin
start_transaction;

(* copy a blob to another database by retrieving it in a blob for *)

for t in atlas.tourism
begin
  store new in guide.tourism using
    new.state := t.state;
    new.city := t.city;
            new.zip := t.zip;
    create_blob n_guide in new.guidebook;
    for o_guide in t.guidebook
      n_guide.segment := o_guide.segment;
      n_guide.length := o_guide.length;
      put_segment n_guide;
    end_for;
    close_blob n_guide;
  end_store;
end;
end_for;
commit;
finish;
end.
```

SEE ALSO

For some guidance on the best approach to processing blobs, see Chapter 6 in this manual. See also the entries in this chapter for:

- **on_error**
- **open_blob**
- **close_blob**

DIAGNOSTICS

See Chapter 8 for a discussion of errors and error handling.

NAME

ready −open database

SYNTAX

ready { *dbhandle-commalist* | *runtime-file* } [*on-error*]

dbhandle ::= { *database-handle* | *runtime-file* **as** *database-handle* }

runtime-file ::= { *database-filespec* | *host-variable* }

on-error ::= **on_error** *statement...* **end_error**

DESCRIPTION

The **ready** statement opens one or more databases for access. When it encounters a **ready** statement,

- Initializes itself internally. The initialization sets up data structures and allocates dynamic memory.
- Looks at the file name of the database and determines if the file is stored on the originating node (a *local database*) or on another node (a *remote database*). provides transparent access to remote databases.
- Opens the database file and looks at the header page. Assuming that the header page identifies the file as containing a valid, unbroken database with the correct version of the on-disk structure, permits further access. Otherwise, it returns an error.

Depending on the switches you set when preprocessing the program with **gpre**, you may not have to issue a **ready** statement to access a database. By default, **gpre** generates a **ready** if one is needed so that the database is automatically readied the first time your program refers to that database. However, if you specify the **manual** switch when you preprocess the program, **gpre** does not generate **ready** (and **start_transaction**) statements. The advantage to using the **manual** switch is that preprocessed code is smaller and simpler.

Finally, you should close each database with a **finish** statement when you are done with it. This practice saves system resources.

ARGUMENTS

dbhandle References either a database assigned a handle in a **database** declaration or a database you specify with *database-filespec* and to which you assign a database handle. The *database-filespec* must be a quoted file specification or a logical name that resolves to a quoted file specification.

In the case of a handle assigned in a previous **database** declaration, the database you ready for runtime access is the same as the one you declared for compiletime access.

The file specification can contain the full pathname, including the name of the node on which the database is stored. If you are in a directory other than the one that contains the database file, the file specification

must include the pathname. If the database is on another node, the *filespec* must include the node name and pathname. You can define a link or logical name for the database file.

File specifications for remote databases have the following form:

Syntax: Remote Database File Specification

VMS to ULTRIX:
node-name::*filespec*

VMS to non-VMS and non-ULTRIX:
node-name^filespec

Within Apollo DOMAIN:
//*node-name*/*filespec*

All Else:
node-name:*filespec*

Make sure that what follows the colon is a valid file specification on the target system; use brackets, slashes, and spaces as appropriate.

runtime-file Readies the named database file. You can use this option if your program accesses only one database.

The file specification can contain the full pathname, including the name of the node on which the database is stored. See the discussion of *dbhandle* for information about accessing remote databases.

on-error Specifies the action to be performed if an error occurs during the ready operation.

EXAMPLES

The following sequence declares a database and readies it:

```
. gdml_156a.epas
        program atlas (input_output);
        database atlas = filename 'atlas.gdb';

        begin

        ready atlas;
        start_transaction;
            .
            .
            .
        rollback;
```

```
        finish atlas;
        end.
```

Another option is to assign the database handle in the **ready** statement. For example, the following sequence declares a compiletime database and readies different databases for runtime access:

```
. gdml_156a.epas
        program ski_areas (input_output);

        database atlas = filename 'atlas.gdb';

        var
          filename  : array [1..40] of char;
          open_database  : boolean;

        begin
          repeat
          begin
            open_database := true;
            write ('Please enter pathname of database ("quit" to exit): ');
            readln (filename);
            if filename = 'quit' then
            begin
              writeln ('Toodles, kid!');
              return;
            end;
            ready filename as atlas
            on_error
            begin
              writeln ('Error during database open.  Status follows.');
              gds_$print_status (gds_$status);
              writeln;
              open_database := false;
            end;
            end_error;
          end;
          until open_database;

        (* do work *)
        finish;

        end.
```

SEE ALSO

See the entries in this chapter for:

- **on_error**
- **database**
- **finish**
- **gpre**

DIAGNOSTICS

See Chapter 8 for a discussion of errors and error handling.

NAME

release_requests −release resources

SYNTAX

release_requests [[**for**] *database-handle*] [*on-error*]

on-error ::= **on_error** *statement...* **end_error**

DESCRIPTION

The **release_requests** statement frees the memory used by the execution tree of all compiled requests for a database and sets the request handles to null.

In most programs, the program logic involves loops that can re-use requests. Therefore, saves requests in their compiled and optimized form. However, if your program finishes and re-readies databases, requests must be re-compiled. The **finish** statement automatically marks requests from that module as obsolete and ensures that they will be re-compiled when, and if, they are re-used.

Large programs consisting of separately compiled modules sometimes have requests in modules that do not contain a **finish** statement. In those cases, you can use the **release_requests** statement to release resources and ensure re-compilation. You must include the **release_requests** statement in one externally callable subroutine in each module that contains a database request. Before you execute a **finish** statement, call each of the ‘‘release’’ subroutines to release resources allocated in its module.

ARGUMENTS

database-handle Specifies the database whose requests you want to release. If you do not specify a database handle, the database software releases requests associated with all open databases.

on-error Specifies the action to be performed if an error occurs during the **for** loop.

EXAMPLES

The following program calls one external routine to perform an action and another to release resources associated with the request:

```
. gdml_160a.epas
        program driver (input, output);
        database atlas = filename "atlas.gdb";

        procedure worker; EXTERN;
        procedure worker_release; EXTERN;

        var  quit  :  array [1..4] of char;

        begin
```

```
repeat
begin
  ready atlas;
  worker;
  worker_release;
  finish;
  write ('Done yet ("yes" to stop): ');
  readln (quit);
end
until quit = 'yes';
end.
```

The following module is called by the preceding program:

```
. gdml_160b.epas
module  worker;
database atlas = EXTERN filename 'atlas.gdb';

procedure worker_release;
begin
  release_requests;
end;


procedure worker;
var
  i  : integer;
begin
  i := 0;
  start_transaction;
  for s in states
    i := i + 1;
  end_for;
  commit;
  writeln ('There are ', i, ' states');
end;
```

SEE ALSO

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

DIAGNOSTICS

See Chapter 8 for a discussion of errors and error handling.

NAME

request-option −request and variable selection

SYNTAX

request-option ::= (*option-commalist*)

option ::= { **level** *integer-expression* |
transaction_handle *host-variable* |
request_handle *host-variable* }

DESCRIPTION

The *request-option* is an optional clause that lets you specify the instantiation (recursion) level of a request, transaction, or request itself that will be affected by the statement.

does not support recursion; if you are using statements in your program, do not involve any operations in a recursive request.

ARGUMENTS

level *integer-expression* Specifies the instantiation level of a request.

transaction_handle *host-variable* Specifies a transaction handle for the transaction in which the statement executes.

request_handle *host-variable* Specifies a request handle for the request in which the statement executes.

EXAMPLES

The following program produces a horizontal organization chart with the president at the top left and the rest of the company moving to the right:

```
. gdml_161a.epas
        program map (input_output);

        database db = filename 'emp.gdb';

        type
          badge_type  = based on employees.badge;
        var
          level  : integer;
          blanks  : array [1..40] of char := [* of ' '];

        procedure  print_next (lev : integer; super : badge_type );
        var
```

```
  offset  : integer;
begin
  for (level lev) e in employees with e.supervisor = super
    offset := (lev) * 4;
    writeln (blanks : offset,
  '....', e.first_name : -1,
  ' ', e.last_name);
    print_next (lev+1, e.badge);
  end_for;
end;    { procedure print next }

begin
        ready;
  start_transaction;
   writeln ('         Employee Roster');
  writeln;
  for e in employees with e.supervisor missing
    writeln (e.first_name : -1, ' ', e.last_name);
    print_next (0, e.badge);
  end_for;
  commit;
  finish;
end.
```

The following program starts a named **consistency** mode transaction to update the BADGE relation:

```
. gdml_164a.epas
       program get_badge (input, output);
       database emp = filename 'emp.gdb';

       var
         get_badge_tr  : gds_$handle;
       begin
              get_badge_tr := nil;
         start_transaction get_badge_tr
           consistency read_write reserving
           badge_num for protected write;
         for (transaction_handle get_badge_tr) b in badge_num
           get_badge := b.badge;
           modify b using
             b.badge := b.badge + 1;
           end_modify;
         end_for;
```

```
  commit get_badge_tr;
end.
```

The following program hires everybody's offspring and assigns them new badge numbers. Note that each request (that is, each **for** and **store**) must use the same request options, even though they are nested. The **modify** statement is not a separate request and does not require a transaction handle. The outer **for** statement is in the default transaction so that it will not read the newly stored records and start prompting for employee grandchildren.

```
. gdml_137a.epas
        program nested_for (input, output);
        database db = filename 'emp.gdb';

        var
          update_tr  : gds_$handle := nil;
          check    : char;
          fnl, lnl  : integer;

        begin
        ready;
        start_transaction update_tr consistency read_write reserving
          badge_num, employees for protected write;

        start_transaction;

        for e in employees
          fnl := 1;
          while (e.first_name [fnl] <>  ' ') do
            fnl := fnl + 1;
          lnl := 1;
          while (e.last_name [lnl] <>  ' ') do
            lnl := lnl + 1;
          write ('Should we hire ', e.first_name:fnl, e.last_name:lnl-1,
         "'s kid? ');
          readln (check);
          if (check = 'y') or (check = 'Y') then
          begin
            for (transaction_handle update_tr) b in badge_num
            begin
              store (transaction_handle update_tr) n_e in employees using
              begin
                write ('What"s the kid"s first name? ');
                readln (n_e.first_name);
                n_e.last_name := e.last_name;
```

```
            write ('What"s the kid"s date of birth? ');
            readln (n_e.birth_date.char[20]);
            n_e.badge := b.badge + 1;
            n_e.department := 'NEP';
            n_e.supervisor := 13;
          end;
          end_store;
          modify b using
            b.badge := b.badge + 1;
          end_modify;
        end;
        end_for;
      end;
    end_for;
    commit update_tr;
    commit;
    finish;
    end.
```

SEE ALSO

See the entries in this chapter for:

- **transaction_handle**
- **for**,
- **start_stream**
- **store**

DIAGNOSTICS

See Chapter 8 for a discussion of errors and error handling.

NAME

rollback −undo changes made during transaction

SYNTAX

rollback [*transaction-handle*] [*on-error*]

on-error ::= **on_error** *statement...* **end_error**

DESCRIPTION

The **rollback** statement restores the database to its state prior to the current transaction. It affects all databases in the transaction, discarding all modified buffers and closing any open record streams.

The **rollback** statement ends a transaction and undoes all changes made to the database since the most recent **start_transaction** statement or since the start of the transaction specified by the transaction handle.

ARGUMENTS

transaction-handle Specifies the transaction you want to roll back. If the transaction you want to roll back has a transaction handle associated with it, you must use that handle when you roll back the transaction.

If you do not specify a transaction handle on a **rollback** statement, commits the ‘‘default’’ transaction. The default transaction is what gets started when you use a **start_transaction** statement without a handle.

on-error Specifies the action to be performed if an error occurs during the rollback operation.

EXAMPLES

The following statements modify the BADGE relation, but rollback the transaction if there is an error:

```
. gdml_166a.epas
        for (transaction_handle get_badge_tr) b in badge_num
          get_badge := b.badge;
          modify b using
            b.badge := b.badge + 1;
          end_modify on_error
            if gds_$status [2] = gds_$deadlock then
              get_badge := 0
             else get_badge := -1;
            rollback get_badge_tr;
            return;
           end_error;
        end_for;
```

SEE ALSO

See the entries in this chapter for:

- **start_transaction**
- *transaction-handle*
- **on_error**

DIAGNOSTICS

A **rollback** statement cannot fail.

NAME

rse −search condition and other activities

SYNTAX

[*first-clause*] *record-source* [*with-clause*] [*reduced-clause*] [*sorted-clause*]

record-source ::= { *relation-clause* | *cross-source* }

relation-clause ::= [*context-variable in*] *relation-name*

cross-source ::= *relation-clause* **cross** *record-source*

DESCRIPTION

The *rse* (record selection expression) clause specifies the search and delivery conditions for record retrieval.

ARGUMENTS

first-clause Limits the records in a stream to the number you specify with an integer. The format of the *first-clause* follows:

Syntax: first-clause of RSE

first *integer*

Any fractional portion of the integer is truncated. Unless you sort the record stream when you use the *first-clause*, *integer* random records are returned.

relation-clause Identifies the target relation. The format of the *relation-clause* follows:

Syntax: relation-clause of RSE

context-variable **in** [*database-handle.*]*relation-name*

The context variable is used for name recognition, and is associated with a relation. A context variable can contain up to 31 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 context variable. For example, it treats **B** and **b** as the same character. For C programs, you can control case sensitivity of context variables with the **either_case** switch when you preprocess your program.

The optional *database-handle* identifies the database for multiple database access.

cross-clause Performs a join operation. The format of the *cross-clause* follows:

Syntax: relation-clause of RSE

cross *relation-clause* [**over** *field-name-commalist*]

The *cross-clause* joins records from two or more different relations in the same database. The relationship can be based on the equality of common fields (equijoin), inequalities (non-equijoin), or where no relationship exists (cross product). Unlike most other clauses of the record selection expression, the *cross-clause* can be repeated to include as many relations as are necessary.

The **over** clause is semantically equivalent to a *with-clause* that equates a field in one relation with a field in another. The *field-name* must be exactly the same in both relations. Otherwise, you must use the *with-clause*, even if both fields are based on the same field.

with-clause Specifies a search condition or combination of search conditions. The format of the *with-clause* follows:

Syntax: with-clause of RSE

with *boolean-expression*

When you pass the search conditions to the access method, it evaluates the condition for each record that might possibly 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 are in the relationship indicated by the operator you specified (for example, equals), the search condition evaluates to ‘‘true’’ and that record becomes part of the record stream. The search condition can result in a value of ‘‘true,’’ ‘‘false,’’ or ‘‘missing’’ for each record.

reduced-clause Performs a project operation, retrieving only the unique values for a field. The format of the *reduced-clause* follows:

Syntax: reduced-clause of RSE

reduced [**to**] *dbfield-expression-commalist*

dbfield-expression ::= [*context-variable.*]*field-name*

When you ask for a record stream projected on a field, the access method considers a list of fields and eliminates records that do not have a unique combination of values for the listed fields.

When you reduce a record stream, you an only reference fields that were mentioned in the **reduced** clause.

sorted-clause Orders the output, returning the record stream sorted by the values of one or more sort keys. The format of the *sorted-clause* follows:

Syntax: sorted-clause of RSE

> **sorted** [**by**] *sort-key-commalist*
>
> *sort-key* ::= [**ascending** | **descending**] *dbfield-expression*
>
> *dbfield-expression* ::= *context-variable.field-name*

You can sort a record stream alphabetically, numerically, by date, and by any combination of these. The *sort-clause* lets you have as many sort keys as you want. Generally speaking, the greater the number of sort keys, the longer it takes for the database software to execute the query.

Each sort key can specify whether the sorting order of the sort key is **ascending** (the default order for the first sort key) or **descending**. The sorting order is ‘‘sticky’’; that is, if you do not specify whether a particular sort key is **ascending** or **descending**, **gpre** assumes that you want the order specified for the most recent key. Therefore, if you list several sort keys, but only include the keyword **descending** for the first key, sorts all keys in descending order.

EXAMPLES

The following query uses a *first-clause*, a *relation-clause*, and a *sorted-clause* to display the two ‘‘youngest’’ states:

```
for first 2 s in states sorted by descending s.statehood
          writeln (s.state_name |
          ’  was admitted to the Union on ’ | s.statehood);
         end_for;
```

The following query uses two *relation-clause* and a *cross-clause* to list a ski area, city, and state in which it is located:

```
for s in states cross ski in ski_areas over state
          writeln (ski.name, ski.city, s.state_name);
         end_for;
```

The following query does the same thing as the preceding query, but uses an explicitly qualified join condition in place of the **cross** shortcut:

```
for s in states cross ski in ski_areas with s.state = ski.state
          writeln (ski.name, ski.city, s.state_name);
         end_for;
```

The following query uses a *reduced-clause* to list the states in which there are ski areas:

```
for ski in ski_areas reduced to ski.state
          writeln (ski.state);
        end_for;
```

The following query uses a *with-clause* to limit the display cities in Texas for which the value of the POPULATION field is not missing:

```
for c in cities with c.state = 'TX' and c.population not missing
          writeln (c.city, c.population, c.altitude);
        end_for;
```

The following statement displays the names of cities that are larger than the capitols of their states:

```
. gdml_171a.epas
        for s in states cross c in cities over state cross
          cs in cities with cs.state = c.state and
          cs.city = s.capitol and
          cs.population < c.population
          sorted by s.state, c.city
          writeln (c.city, s.state_name, ' is larger than ', s.capitol);
        end_for;
```

The following statement displays only the names of states in which the capitol is not the largest city:

```
for s in states cross c in cities over state cross
          cs in cities with cs.state = c.state and
          cs.city = s.capitol and
          cs.population < c.population
          sorted by s.state
          reduced to s.state, s.capitol
          writeln (s.state_name, ' contains cities larger than ', s.capitol);
          end_for;
        commit;
        finish;
        end.
```

SEE ALSO

See the entries in this chapter for:

- *boolean-expression*

- *value-expression*

DIAGNOSTICS

See Chapter 8 for a discussion of errors and error handling.

NAME

start_stream −create record stream

SYNTAX

start_stream [*request-option*] *stream-name*
using *rse* [*on-error*]
statement...
end_stream *stream-name* [*on-error*]

on-error ::= **on_error** *statement...* **end_error**

DESCRIPTION

The **start_stream** statement declares and opens a record stream.

You can start a stream with the **for** statement or with the **start_stream** statement. The **for** statement is generally recommended. However, you may want to use a **start_stream** statement if you are:

- Processing several streams in parallel.
- Processing a stream until some condition is met, and then exiting from the stream.

ARGUMENTS

request-option Specifies a transaction handle and/or request handle that determine the transaction and/or request in which the **start_stream** statement executes.

stream-name Names the stream. The name can contain up to 31 alphanumeric characters, dollar signs ($), and underscores (_).

The context of the stream name is the whole module that contains the **start_stream** statement, so you cannot re-use a stream name in the same module.

rse Specifies the record selection criteria used to create the record stream.

on-error Specifies the action to be performed if an error occurs when you start the stream or when it terminates. Errors on the **end_stream** generally occur only in extreme cases, such as a network partition while the stream is still open.

statement Specifies or host language statements to be executed within the stream. The statements you include are subject to the following rules:

- If you include more than one *statement*, you must separate them using the host language convention.
- If you use other statements while the stream is open, those statements can use only the context variables declared in the **start_stream**/**end_stream** block, in outer blocks, or in

inner blocks. You can re-use the context variables outside those blocks.

EXAMPLE

The following program illustrates the use of the **start_stream** statement in a loop that may be terminated by user interaction:

```
. gdml_173a.epas
        program map (input_output);

        database db = filename 'atlas.gdb';
        var  end_of_stream  : boolean;
          genug    : char;

        begin
          start_stream geodata using c in cities
            sorted by c.latitude, c.longitude;
          end_of_stream := false;
          while not end_of_stream do begin
            fetch geodata
              at end end_of_stream := true;
            end_fetch;
            if not end_of_stream then
            begin
              writeln (c.latitude, c.longitude, c.altitude,
                c.city, c.state);
              write ('Seen enough? (Y/N) ');
              readln (genug);
              if genug = 'Y' then
                end_of_stream := true;
            end;
          end;
          end_stream geodata;
          commit;
          finish;
        end.
```

SEE ALSO

See the entries in this chapter for:

- *request-option*
- *rse*
- **on_error**

DIAGNOSTICS

See Chapter 8 for a discussion of errors and error handling.

NAME

start_transaction −begin transaction

SYNTAX

start_transaction [*transaction-handle*]
[**concurrency** | **consistency**]
[**read_write** | **read_only**]
[**wait** | **nowait**]
[*reserving-clause*]
[*on-error*]

reserving-clause ::= **reserving** *reserved-relation-commalist*

reserved-relation ::= [*database-handle.*]*relation-name*
for [**protected**] { **read** | **write** }

on-error ::= **on_error** *statement...* **end_error**

DESCRIPTION

The **start_transaction** statement begins a group of statements that are executed as one logical unit.

A process can start any number of independent transactions. This capability facilitates the development of server processes and allows system service routines to use databases without affecting user-level database activity.

ARGUMENTS

transaction-handle Declares a name that you can use when you have to reference multiple transactions in a program.

If you start a transaction without specifying a transaction handle, **gpre** starts the ‘‘default transaction.’’ There is one default transaction per process. When **gpre** encounters a subsequent statement without a transaction handle, it generates a test for the default handle. If there is no default transaction, **gpre** starts one. In any case, **gpre** applies statements without transaction handles to the default transaction.

concurrency (default)
consistency The **concurrency** default provides high throughput and concurrency with generally satisfactory consistency. No transaction sees any data written by another active transaction.

The **consistency** option provides a high level of database consistency that guarantees that all transactions are serializable (that is, having the same effect on the database as if all transactions were run sequentially in some order) at the expense fo concurrency.

To ensure a deadlock-free transaction, use the **consistency** option and reserve the relations required by the

transaction for **read** or **write** depending on the mode in which they will be used. However, this option does not allow concurrent access to the reserved relations.

See Chapter 5 for more information about **concurrency** and **consistency**.

read_write (default)
read_only The default intention of a transaction is that it will read and write data. You may choose to declare a transaction **read_only** to document its behavior or as a check on program logic.

wait (default)
nowait The default action if your program encounters a locked object is to wait until the lock goes away. The **nowait** option produces a *lock_conflict* error whenever a program encounters a locked object. The **nowait** option is not recommended because it requires more error handling in a program and can lead to unnecessary rollbacks.

reserving Lists the relations to be used in the transaction. locks those relations for your access if you choose **consistency** mode. You must list each relation that the transaction will ‘‘touch’’ (that is, if it is used at all, in any capacity). List relations individually; you can specify different relation locking criteria for each. However, if you choose **read_only** for the transaction (see above), you cannot reserve a relation for **write**.

If you have a **concurrency** mode transaction, you can optionally reserve a relation for **protected write**. This mode allows other users to read the relation, but prevents them from writing to it. By default, **concurrency** mode transactions are reserved for shared access, an access mode that all users write to the relation.

The **protected write** reserving option is the default for **consistency** mode transactions.

To ensure a deadlock-free transaction, use the **consistency** option and reserve the relations required by the transaction for **read** or **write** depending on the mode in which they will be used.

on-error Specifies the action to be performed if an error occurs when you start the transaction.

EXAMPLE

The following statement starts a transaction that will become the default transaction because there is no transaction handle:

```
start_transaction;
```

The following statement starts a transaction and assigns a transaction handle:

```
program zip_update (input_output);
        database db = filename 'atlas.gdb';
        var
```

```
  zippity_doo_dah : gds_$handle  := nil;
begin
. {startran_2.epas}
start_transaction zippity_doo_dah;
    .
    .
    .
commit zippity_doo_dah;
finish;
end.
```

The following statement starts a transaction with a **reserving** clause:

```
program zip_update (input_output);
        database db = filename 'atlas.gdb';
        var
          zippity_doo_dah : gds_$handle  := nil;
        begin

        start_transaction zippity_doo_dah
          read_write consistency
          reserving catalog.catalog_items for write;
```

SEE ALSO

See the entries in this chapter for:

- **prepare**
- **commit**
- **rollback**
- **on_error**

DIAGNOSTICS

See Chapter 8 for a discussion of errors and error handling.

NAME

store −insert new record

SYNTAX

store [*request-option*] *relation-clause* **using**
statement...
end_store [*on-error*]

on-error ::= **on_error** *statement...* **end_error**

DESCRIPTION

The **store** statement inserts a new record into a relation.

You cannot store records into views formed from more than a single relation. Rather, you must store them into the source relations.

ARGUMENTS

request-option Specifies a transaction handle and/or request handle that determine the transaction and/or request in which the **store** statement executes.

If you nest a **store** statement inside a **for** loop and use an explicit transaction handle on the **for** statement, you must also use the transaction handle on the **store** statement. Otherwise, the **store** statement will be executed inside the default transaction.

relation-clause Specifies the relation into which the new record is to be stored. See the entry for *rse* in this chapter for more information about the *relation-clause*.

on-error Specifies the action to be performed if an error occurs during the store operation.

statement Specifies the action to be taken in storing the record(s). The *statement*s are typically assignments. If you include more than one *statement*, you must separate them using the host language convention.

EXAMPLE

The following statement stores a new record in SKI_AREAS:

```
store ski in ski_areas using
         write ('Name: ');
         readln (ski.name);
         write ('City: ');
         readln (ski.city);
         write ('State: ');
```

```
          readln (ski.state);
          write ('Type: ');
          readln (ski.type);
        end_store;
```

The following statements use an outer **for** loop to create a record stream from which a **store** statement takes some values, host variables supply some values, and unreferenced fields are set to missing:

```
for oldcity in cities with oldcity.city_name = hostvar1
          store newcity in cities using
            newcity.city = hostvar2;
            newcity.state = oldcity.state;
            newcity.population = oldcity.population * hostvar3;
            newcity.altitude = oldcity.altitude;
          end_store;
        end_for;
```

The following program hires everybody's offspring and assigns them new badge numbers. Note that each request (that is, each **for** and **store**) must use the same request options, even though they are nested. The **modify** statement is not a separate request and does not require a transaction handle. The outer **for** statement is in the default transaction so that it will not read the newly stored records and start prompting for employee grandchildren.

```
. gdml_137a.epas
        program nested_for (input, output);
        database db = filename 'emp.gdb';

        var
          update_tr  : gds_$handle := nil;
          check    : char;
          fnl, lnl  : integer;

        begin
        ready;
        start_transaction update_tr consistency read_write reserving
          badge_num, employees for protected write;

        start_transaction;

        for e in employees
          fnl := 1;
          while (e.first_name [fnl] <>  ' ') do
            fnl := fnl + 1;
          lnl := 1;
```

```
  while (e.last_name [lnl] <>  ' ') do
    lnl := lnl + 1;
  write ('Should we hire ', e.first_name:fnl, e.last_name:lnl-1,
'"s kid? ');
  readln (check);
  if (check = 'y') or (check = 'Y') then
  begin
    for (transaction_handle update_tr) b in badge_num
    begin
      store (transaction_handle update_tr) n_e in employees using
      begin
        write ('What"s the kid"s first name? ');
        readln (n_e.first_name);
        n_e.last_name := e.last_name;
        write ('What"s the kid"s date of birth? ');
        readln (n_e.birth_date.char[20]);
        n_e.badge := b.badge + 1;
        n_e.department := 'NEP';
        n_e.supervisor := 13;
      end;
      end_store;
      modify b using
        b.badge := b.badge + 1;
      end_modify;
    end;
    end_for;
  end;
end_for;
commit update_tr;
commit;
finish;
end.
```

SEE ALSO

See the entries in this chapter for:

- *request-option*
- *rse*
- **on_error**

DIAGNOSTICS

See Chapter 8 for a discussion of errors and error handling.

NAME

transaction-handle −identify transaction

SYNTAX

host-variable

DESCRIPTION

The *transaction-handle* clause specifies the name of a transaction in several statements.

If you do not start a transaction with the **start_transaction** statement, choosing instead to let **gpre** start transactions as needed, you can still specify the transaction under which you want a statement to be executed by declaring a transaction handle in the *request-option* clause of the **for**, **store**, and **start_stream** statements. If that transaction does not exist, **gpre** starts it.

ARGUMENTS

host-variable A host language program variable that serves as the transaction handle.

- For BASIC, the transaction handle must be declared as LONG and set to 0.
- For C programs, the transaction handle must be declared as a long integer initialized to **null** (0).
- For COBOL, the transaction handle must be declared as PIC S(9) COMP.
- For FORTRAN programs, the transaction handle must be declared as I*4 set to 0.
- For Pascal programs, the transaction handle must be explicitly declared in the program as a pointer to any type and initialized to **nil** before use. The variable **gds_$handle** is pre-declared as a type for Pascal.
- For PL/I, the transaction handle must be declared as a pointer and initialized to **NULL()**.

EXAMPLE

The following Pascal example starts two named transactions, performs some unspecified data manipulation in each, then writes the changes for only the specified transaction to the database, and continues with the other transaction committing it:

```
start_transaction store_resort;
        start_transaction drop_resort;
          .
          .
          .
        for (transaction_handle store_resort)
          .
          .
```

```
  .
for (transaction_handle drop_resort)
  .
  .
  .
commit store_resort;
  .
  .
  .
commit drop_resort;
```

SEE ALSO

See the entries in this chapter for:

- **commit**
- **prepare**
- **rollback**
- **start_transaction**
- **for**
- **store**
- **start_stream**
- *request-option*

DIAGNOSTICS

See Chapter 8 for a discussion of errors and error handling.

NAME

value-expression −calculating value

SYNTAX

value-expression ::= { *arithmetic-expression* | *dbfield-expression* | *numeric-literal-expression* | *quoted-string-expression* | *(value-expression*) | - *value-expression* }

DESCRIPTION

The *value-expression* is a symbol or string of symbols from calculates a value. uses the result of the expression when executing the statement in which the expression appears.

ARGUMENTS

arithmetic-expression Combines value expressions and arithmetic operators. The format of the *arithmetic-expression* follows:

Syntax: arithmetic-expression Value Expression

value-expression-1 { + | - | * | / | } *value-expression-2*

You can add (+), subtract (-), multiply (*), and divide (/) value expressions in record selection expressions. Arithmetic operators are evaluated in the normal order. Use parentheses to change the order of evaluation.

dbfield-expression References database fields. This expression can occur in several clauses of *rse* and *boolean-expression*. The format of the *dbfield-expression* follows:

Syntax: dbfield-expression Value Expression

context-variable.field-name[**.null** | *.datatype*]

The *context-variable* qualifies the database field for multi-relation operations. Declare the context variable for a relation in the *relation-clause* of the record selection expression.

The optional **.null** qualifier allows access to the null flag for the field. If you reference the null flag in a **store** or **modify** statement, you must set it explicitly. If the null flag remains true (that is, non-zero), the field will be stored as missing even if you supply a value.

The optional *.datatype* qualifier lets you ‘‘cast’’ a database field with a datatype other than that with which it is stored. **Gpre** automatically takes care of datatype conversion, but you can ‘‘convert’’ a field for the

duration of a request to the datatype of your choice. Chapter 4 discusses casting in more detail and lists the datatype conversions supported by casting.

numeric-literal-expression Represents a decimal number as a string of digits with an optional decimal point. The format of the *numeric-literal-expression* follows:

Syntax: numeric-literal-expression Value Expression

string[*.string*]

quoted-string-expression A string of ASCII characters enclosed in single (’) or double (") quotation marks, depending on host language requirements. The format of the *quoted-string-expression* follows:

Syntax: quoted-string-expression Value Expression

"string"

ASCII printing characters are:

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

EXAMPLES

The following statement uses *dbfield-expression*s to display the city and state, an *arithmetic-expression* that calculates and displays the altitude in meters, a *numeric-literal-expression* (0.3048) used in the arithmetic operation, and two *quoted-string-expression*s to anglicize the Pascal **writeln** display:

```
for c in cities cross s in states over state
          writeln (c.city, s.state_name, ’ is situated at ’,
             c.altitude * 0.3048, ’ meters above sea level.’);
        end_for;
```

See any of the other manual pages in this chapter for examples of the value expression.

SEE ALSO

See the entries in this chapter for:

- *boolean-expression*

- *rse*

DIAGNOSTICS

See Chapter 8 for a discussion of errors and error handling.