NAME

define database −create a database

SYNTAX

define database *quoted-filespec* [**{** *textual commentary* **}**] [**security_class** *class-name*];

DESCRIPTION

The **define database** statement provides the name for the database to be created.

The **define database** statement must be the first statement in the source file or input to **gdef**.

ARGUMENTS

quoted-filespec A valid file specification enclosed in single (’) or double (") quotation marks. If the shell you regularly use is case-sensitive, make sure that you always reference the database file exactly as it is spelled in the **define database** statement.

The file specification can contain the full pathname to another node in the network. 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*

{ *textual commentary* **}** Stores the bracketed comments about the database in the database. The commentary can include any of the following ASCII characters:

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

class-name Associates a security class with the database. See the entry for **define security_class** in this chapter for more information.

EXAMPLE

The following statements each appear as the first statement in a source file used to define a database:

```
define database "/gds/examples/atlas.gdb"
        .sp
        define database "atlas.gdb";
        .sp
        define database "/usr/jimbo/boats.gdb"
          { the ubiquitous test database }
          security_class lmu;
        .sp
        define database "/usr/igor/datafiles/atlas.gdb";
```

SEE ALSO

See Chapter 4 in this manual. See also the entries in this chapter for:

- **gdef**
- **define security_class**

DIAGNOSTICS

See Chapter 3 for a discussion of errors and error handling.

NAME

define field −define a field

SYNTAX

define field *field-name datatype field-attributes*;

DESCRIPTION

The **define field** statement describes the characteristics of a new field for later inclusion in a relation.

You can also define fields in relations. See the entries for **define relation** and **modify relation** in this chapter for more information.

To disallow duplicate values for a field, define an index for the field in the relation and use the **unique** option. See the entry for **define index** in this chapter for more information about defining indexes.

ARGUMENTS

field-name Names the field you want to create. A field name can contain up to 31 alphanumeric characters, dollar signs ($), and underscores (_). However, it must start with an alphabetic character.

datatype Specifies the field's datatype. The datatype specification must precede other field attributes and descriptive comments. For detailed information about supported datatypes, see the entry for *field-attributes* in this chapter.

field-attributes Specifies the length or scale if appropriate, and several optional field characteristics. You can include a textual description of the field, an explicit missing value, a query name, and a validation expression.

For detailed information about optional attributes, see the entry for *field-attributes* in this chapter.

EXAMPLE

The following statements define fields:

```
. tcs?
        define field tolerance long scale -2;
        .sp
        define field blurb blob sub_type text
          stream segment_length 60
          { text for catalogue article };
        .sp
        define field manufacturer char[10]
          valid if
            (manufacturer ne "SLEAZOLA" and
            manufacturer ne "SHODTECH" and
```

```
    manufacturer ne "SCHLOKHAUS")
    { add bad suppliers as necessary }
  missing_value is "N/A";
.sp
define field price float
  valid if
    (price > 0 or
    price missing)
  missing_value is -1.99;
.sp
define field part_number char[5];
```

SEE ALSO

See Chapters 4 and 5 in this manual. See also the entries in this chapter for:

- *field-attributes*
- **define relation**
- **modify relation**

DIAGNOSTICS

See Chapter 3 for a discussion of errors and error handling.

NAME

define index −create an index

SYNTAX

define index *index-name* [**for**] *relation-name*
[**unique**] [**{** *textual-commentary* **}**]
field-name-commalist;

DESCRIPTION

The **define index** statement defines an index for a relation. You must define a relation before you can create an index for it.

automatically maintains all indexes. You do not have to reference an index when you access data—the access method does it automatically.

ARGUMENTS

index-name Names the index you want to create. An index name can contain up to 31 alphanumeric characters, dollar signs ($), and underscores (_). However, it must start with an alphabetic character.

relation-name Specifies the relation for which you are defining the index.

unique Disallows duplicate values in the index.

Try to index on fields used as *primary keys*, such as unique identification numbers, part numbers, employee numbers, social security numbers (although social security numbers are not reliably unique and, by law, should not be used for identification purposes), and so on. You can define a unique index by specifying the optional keyword **unique**. If you do so, the values for *field-name* or combinations of *field-name*s must then be unique. If you try to store a value that already exists, the assignment operation fails.

If you create a multi-segment index, you should first consider which of the key fields is likely to have the unique values. Having done so, you should list the *field-name*s in descending order by uniqueness Such ordering improves index compression.

{ *textual-commentary* **}** Stores the bracketed comments about the relation in the database. The commentary can include any of the following ASCII characters:

- Uppercase alphabetic: *A—Z*
- Lowercase alphabetic: *a—z*
- Numerals: *0—9*
- Blanks, tabs, and carriage returns

- Special characters: ! @ # $ % ^ & * ( ) _ - + = ‘ ~ [ ] < > ; : ’ " \ | / ? . ,

field-name Specifies one or more fields from *relation-name* that will be indexed.

You can create a single or multi-segment index for a relation. A single-segment index consists of a single field, while a multi-segment index consists of two or more fields. In both cases, you should avoid indexing a field that has few unique values. Such indexes provide little performance improvement and can reduce update performance. Finally, because of the nature of the blob datatype, you cannot index a blob field.

EXAMPLE

The following statements define relations and some indexes for them:

```
. tcs?
        define relation states
          .
          .
          .
        define relation cities
          .
          .
          .

        define index state_idx1 for states
          unique state;
        .sp
        define index state_idx2 for states
          unique state, state_name;
        .sp
        define index river_idx1 for rivers
          unique  { speed access and
            eliminate duplicates }
          river;
        .sp
        define index rivstat_idx1 for river_states
          unique river, state;
```

SEE ALSO

See the entries in this chapter for:

- **gdef**
- **delete**

DIAGNOSTICS

See Chapter 3 for a discussion of errors and error handling.

NAME

define relation −define a relation

SYNTAX

define relation *relation-name*
[**{** *textual-commentary* **}**]
[**security_class** *class-name*]
field-description-commalist;

field-description::= { *included-field* | *new-field* | *renamed-field* | *computed-field* }

DESCRIPTION

The **define relation** statement creates a relation. A relation can consist of:

- Included fields. Such fields are defined in previous **define field** or **define relation** statements, and can have optionally specified local attributes. The local attributes are described at length in the entry for *field-attributes* in this chapter.

  Syntax: included-field

  field-name [*local-attributes*]

  local-attributes ::= { *comments* | *edit-string* | *query-name* |
  security-class | **position** *n* }

  If you specify a local attribute that conflicts with the attribute defined in the field definition, the local attribute overrides the global attribute for this use of the field. You cannot override the field's missing value, validation criteria, and datatype and related subclauses (scale, segment length, subtype, and so on) because they are part of the core definition of the global field.

- New fields that are defined in the relation.

Syntax: new-field

field-name field-attributes

local-attributes ::= { *comments* | *edit-string* | *query-name* | *security-class* | **position** *n* }

This clause defines a new field within the relation, instead of using existing or virtual fields. Because the field is defined from scratch, you must include the field's datatype. **Gdef** adds these fields to the global list of fields for that database, thereby making those fields available for inclusion in subsequent relation definitions.

- Fields defined elsewhere and renamed for the relation.

Syntax: renamed-field

local-name **based on** *field-name* [*local-attributes*]

local-attributes ::= { *comments* | *edit-string* | *query-name* | *security-class* | **position** *n* }

This clause renames a field for use in the relation being defined, at the same time retaining all characteristics except those you change explicitly. You can include a textual description, an edit string, a query name, a security class, and a position clause.

Except for the security class and position clauses, these local attributes are described at length in the entry for *field-attributes* in this chapter.

If you specify a local attribute that conflicts with the attribute defined for the global field in the **define field** statement, the local attribute overrides the global attribute. You cannot override the field's missing value, validation criteria, and datatype and its related subclauses (scale, segment length, and so on), because they are part of the core definition of the global field.

- Virtual or computed fields.

Syntax: computed-field

local-name [*datatype*] **computed by (***value-expression***)**

This clause defines a computed or virtual field that consists of a formula rather than a storage location. Note that the value expression must be in parentheses. never stores data in such fields, but it calculates the formula and retrieves requested data. If you do not specify the optional *datatype*, calculates an appropriate datatype.

Because the computed field depends on values from its context, it cannot be used in arbitrary relations. **Gdef** generates a unique name for the global portion of the field. Therefore, a computed field does not need a name that is unique in the database.

ARGUMENTS

relation-name Names the relation you want to create. A relation name can contain up to 31 alphanumeric characters, dollar signs ($), and underscores (_). However, it must start with an alphabetic character.

{ *textual-commentary* **}** Stores the bracketed comments about the relation in the database. The commentary can include any of the following ASCII characters:

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

class-name Associates a security class with the relation or fields within the relation. See the entry for **define security_class** in this chapter.

field-name Specifies the name you want for the field in the relation. The **define relation** statement supports several types of field definitions, all of which you can use in the same relation definition.

position *n* Specifies the position (left to right) that **qli** uses to print when displaying the relation. The first field is position *0*. For example, if there are three fields, A, B, and C, with defined positions of 1, 0, and 2, respectively, **qli** displays these fields in the order B, A, and C.

If you do not specify a position, **qli** uses the order in which the fields are defined or included in the relation.

EXAMPLE

The following example defines several fields with **define field** statements and then includes them in a relation:

```
. tcs?
        define field state char[2];
        define field state_name varying [25];
        define field city varying[25];
        .sp
        define relation states
          { basic information about states }
          state,
          state_name,
          area long,
          statehood char[4],
          capitol based on city;
```

The following statement defines several fields within a relation, defines some computed values, and also includes several existing fields:

```
. tcs?
        define relation cities
          { info about capitols and largest cities }
          city,
          state,
          population long,
          altitude long,
          latitude_degrees varying[3]
            query_name latd,
          latitude_minutes char[2]
            query_name latm,
          latitude_compass char[1]
            query_name latc,
          longitude_degrees varying[3]
            query_name longd,
          longitude_minutes char[2]
            query_name longm,
          longitude_compass char[1]
            query_name longc,
          latitude computed by (
            latitude_degrees | " " |
            latitude_minutes |
            latitude_compass),
          longitude computed by (
            longitude_degrees | " " |
            longitude_minutes |
            longitude_compass);
```

The following example defines a field and then includes it in a relation under different names:

```
. tcs?
        define field population long;
        .sp
        define relation populations
          { US census data by state }
          state,
          census_1950 based on population
            query_name c1950,
          census_1960 based on population
            query_name c1960,
          census_1970 based on population
            query_name c1970,
          census_1980 based on population
            query_name c1980;
```

SEE ALSO

See Chapter 6 in this manual and the entry for *value-expression* in See also the entries in this chapter for:

- *field-attributes*
- **define field**

DIAGNOSTICS

See Chapter 3 for a discussion of errors and error handling.

NAME

define security_class −establish access control

SYNTAX

define security_class *class-name element-commalist*;

element ::= { *grantee* | **view** *view-name* } *privilege-list*

privilege-list ::= { **R** | **W** | **P** | **C** | **D** }

VAX/VMS:
grantee :== **[***uic***]**

UNIX:
grantee :== **[***group, user***]**

APOLLO:
grantee :== *user*[.*project*[.*organization*[.*node*]]]

DESCRIPTION

The **define security_class** statement establishes access control lists that you can associate with databases, relations, views, and fields in relations and views.

ARGUMENTS

class-name Names the security class you want to create. A security class name can contain up to 31 alphanumeric characters, dollar signs ($), and underscores (_). However, it must start with an alphabetic character.

element Defines the access control for individual user or group. You can use the wildcard character % to substitute for any of these identifiers.

The privilege list specifies the following privileges:

- **R** (read). Users with read privilege can read the database, relation, view, or field.
- **W** (write). Users with write privilege can write to the database, relation, view, or field.
- **P** (protect). Users with protect privilege can change the security class for the database, relation, view, or field.
- **C** (control). Users with control privilege can change the metadata for the database, relation, view, or field.
- **D** (delete). Users with delete privilege can delete the definition for the relation.

Gdef automatically orders *element* entries with the most specific appearing first. For example, the APOLLO

element ‘‘%.%.%.%’’ comes last, as does the VMS entry ‘‘[*]’’.

The more general access controls override the more specific. For example, if you have read privilege for a database, but write for a specific relation in that database, you can only read that relation.

Suppose you have a personnel database and you want to hide SALARY information from most people. You can write to a relation in which there is field that you cannot read. To do so, provided you can write to the relation, store a record in the relation without referencing the *verboten* field. The access method automatically sets that field’s value to missing. If you want to prohibit someone from getting around the field-level security in this manner, define the missing value as an invalid value.

EXAMPLE

The following statements define a database, a security class, a field, and a relation, and associates the security class with each of the entities:

```
define database "war_effort.gdb"
          security_class staff
          { This database holds secret and
            top secret information about the
            Union war effort in 1863.  Data protection
            was designed at the specific request
            of President Lincoln };
        .sp
        define security_class cabinet_level
          { this class is used on sensitive data that
            must be shared with the cabinet }
            lincoln.pres.usa pcrwd,
            chase.treasury.cabinet r,
            seward.state.cabinet crw,
            %.%.cabinet rw,
            view sanitized_data r;
        .sp
        define security_class staff
          { used for data that the cabinet is going to leak
            anyway, so why not let everybody at it? }
            lincoln.pres.usa pcrwd,
            %.%.cabinet crw,
            %.%.associates r,
            %.%.staff r;
        .sp
        define security_class top_secret
          { this stuff is so secret even the president
            probably should not see it }
            lincoln.pres.usa pcrwd;
```

```
.sp
define field general char [20];
define field battle char [20];
define field army char [20];
define field force short;
define field opinion blob;
define field location char [30];
define field destination char [30];
.sp
define relation armies
  { who is where, and what the president thinks of them }
    security_class cabinet_level
    army,
    general,
    force,
    location,
    destination security_class top_secret,
    opinion security_class top_secret;
.sp
define view sanitized_data of a in armies
  with a.location not containing "Virginia"
  { semi-public information }
    a.army,
    a.general;
.sp
modify relation rdb$security_classes
    security_class top_secret;
```

SEE ALSO

See Chapter 7 in this manual. See also the entries in this chapter for:

- **define database**
- **define relation**
- **define view**

DIAGNOSTICS

See Chapter 3 for a discussion of errors and error handling.

NAME

define trigger −create integrity check

SYNTAX

define trigger for *relation-name*
[**store:** *trigger-action*]
[**modify:** *trigger-action*]
[**erase:** *trigger-action*]

DESCRIPTION

The **define trigger** statement specifies an action that performs automatically whenever you execute a store, modify, or erase operation on the relation.

The trigger language is based on the **qli** variant of including the **any** and statistical expressions. It differs from the **qli** variant in the following ways:

- The **abort** *n* statement causes the action to terminate with a status of *gds_$integ_failed*. The error message includes the number you supplied in the trigger. If you are handling errors yourself in a program, the error number is the fourth longword in the status vector.
- The trigger language does not include standalone **modify** or **erase** statements. Therefore, you must include such statements in a **for** loop in triggers.
- Context variables are required in the trigger language.

ARGUMENTS

store: *trigger-action*
modify: *trigger-action*
erase: *trigger-action* Specifies that the trigger action is to be performed on a store (or insert), modify (or update), or erase (or delete) operation. Each of these operations can have a separate trigger action.

trigger-action Specifies a statement that executes whenever you store a new record into the relation, modify a field from a record in the relation, or erase a record from the relation. **Gdef** supplies two predefined context variables, **old** and **new**, for use in the trigger action. **Old** refers to the record you are modifying or erasing, and **new** refers to the new record or version you are creating.

See the for information about data manipulation.

EXAMPLES

The following statements define a relation, a trigger, a second relation that keeps an audit trail of activity on the first relation, and rejects new records without a widget name or modified widgets whose new number is less than the old number:

```
. no_name
        define database "not_yachts.gdb";
        .sp
        define relation widgets
          name char [10],
          number short;
        .sp
        define relation log
          name,
          what char [6],
          old_number based on number,
          new_number based on number,
          when date;
        .sp
        define trigger for widgets
          store:
            store x in log
            if new.name missing abort 1
              x.what = "STORE";
              x.name = new.name;
              x.new_number = new.number;
              x.when = "today";
            end_store;
          modify:
            store x in log
            if new.number missing abort 2
              x.what = "MODIFY";
              x.name = new.name;
              x.old_number = old.number;
              x.new_number = new.number;
              x.when = "today";
            end_store;
          erase:
            store x in log
              x.what = "ERASE";
              x.name = old.name;
              x.old_number = old.number;
              x.when = "today";
            end_store;
        end_trigger;
```

SEE ALSO

See Chapter 8 in this manual.

DIAGNOSTICS

See Chapter 3 for a discussion of errors and error handling.

NAME

define view −create view

SYNTAX

define view *view-name* **of** *rse*
[**{** *textual-commentary* **}**]
[**security_class** *class-name*]
field-name-commalist;

field-name ::= { *included-field* |
renamed-field |
computed-field }

DESCRIPTION

The **define view** statement creates a view definition that can include fields from one or more relations. A view can be:

- A simple vertical subset of a relation. That is, the view limits the fields that are displayed.
- A simple horizontal subset of a relation. That is, the view limits the records that are displayed.
- A single relation subset vertically and horizontally.
- A combination of relations subset horizontally, vertically, or both.

You can access views as if they were relations. That means you can select records from it, project on its fields, join it with another relation or itself, or involve it in a union. However, the source of the view determines which, if any, update operations you can perform on the view:

- If the view is a vertical subset of a single relation, you can treat it as a relation for both retrieval and update purposes, provided that all excluded fields allow missing values.
- If the view references more than one relation, you cannot update records through the view. Instead, you must update the records through their source relations. This restriction avoids update anomalies.

A view can consist of:

- Fields from any of the source relations, with optionally specified local attributes

Syntax: included-field

dbfield-expression [*local-attributes*]
local-attributes ::= { *comments* | *edit-string* | *query-name* |
security-class | **position** *n* }

The *dbfield-expression* is a field name qualified by a context variable. The context variable is declared in the record selection expression used to limit the records for the view.

- Fields from any of the source relations, renamed for use in the view (**from** *dbfield-expression* [*local-attributes*]).

Syntax: renamed-field

local-name **from** *dbfield-expression* [*local-attributes*]
local-attributes ::= { *comments* | *edit-string* | *query-name* |
security-class | **position** *n* }

The **from** clause renames a qualified field name for use in the view being defined, at the same time retaining all characteristics except those you change explicitly.

- Virtual fields (**computed by (***value-expression***)**).

Syntax: computed-field

local-field [*datatype*] **computed by** (*value-expression*) [*local-attributes*] }
local-attributes ::= { *comments* | *edit-string* | *query-name* |
security-class | **position** *n* }

The **computed by** clause defines a virtual field that is a formula rather than a reference to stored fields.

For each of the fields in a view, you can include a textual description, an explicit missing value, a query name, and a security class. Except for *security-class* and the **position** clause, these local attributes are described at length in the entry for *field-attributes* in this chapter.

If you specify a local attribute that conflicts with the attribute defined for the global field in the **define field** or **define relation** statement, the local attribute overrides the global attribute. The only attributes you cannot override are the field's validation criteria, and datatype and related subclauses (scale, segment length, and so on).

ARGUMENTS

view-name Names the view you want to create. A view name can contain up to 31 alphanumeric characters, dollar signs ($), and underscores (_). However, it must start with an alphabetic character.

rse Selects the records that constitute the view. You can use any option of the record selection expression in defining the view except the *first-clause* and the *sorted-clause*.

See for more information on record selection.

{ *textual-commentary* **}** Stores a bracketed descriptive comment about the view. The commentary can include any of the following ASCII characters:

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

security_class *class-name* Associates a security class with the view. You can also treat a view as if it were a user. See the entry **define security_class** in this chapter for more information.

field-name Specifies the field(s) you want to include in the view. The **define view** statement supports several types of field definition, all of which you can use in the same view definition.

position *n* Specifies the position (left to right) that **qli** uses to print when displaying the view. The first field is at position *0*. For example, if there are three fields, A, B, and C, with defined positions of 1, 0, and 2, respectively, **qli** displays these fields in the order B, A, and C.

If you do not specify a position, **qli** uses the order in which the fields are defined or included in the view.

EXAMPLES

The following statement defines a vertical subset of a relation (that is, subset of fields):

```
. tcs?
        define view geo_cities of c in cities
          { subset of CITIES with geographic data only }
          c.city,
          c.state,
          c.altitude,
          c.latitude,
          c.longitude;
```

The following view defines both a horizontal and vertical subset (that is, selected records and a subset of fields):

```
. tcs?
        define view middle_america of c in cities
```

```
  with longitude_degrees beween 79 and 104 and
  latitude_degrees between 33 and 42
  c.city,
  c.state,
  c.altitude;
```

The following view defines a horizontal subset of a relation by using an existential qualifier to test another relation for field values from the first relation:

```
. tcs?
   define view ski_states of s in states
     with any shush_boom in ski_areas
     with s.state = shush_boom.state
       s.state,
       s.capitol,
       s.area,
       s.population;
```

The following view joins two relations, using values from fields in both relations to compute population densities for each decade's census:

```
. tcs?
   define view population_density of p in populations
     cross s in states over state
     p.state,
     density_1950 computed by
       (p.census_1950 / s.area),
     density_1960 computed by
       (p.census_1960 / s.area),
     density_1970 computed by
       (p.census_1970 / s.area),
     density_1980 computed by
       (p.census_1980 / s.area);
```

SEE ALSO

See Chapter 6 in this manual and the entry for *value-expression* in See also the entries in this chapter for:

- *field-attributes*
- **define security_class**

DIAGNOSTICS

See Chapter 3 for a discussion of errors and error handling.

NAME

delete −erase metadata

SYNTAX

delete { **field** *field-name* | **index** *index-name* |
relation *relation-name* | **security_class** *class-name* |
trigger for *relation-name* | **view** *view-name* };

DESCRIPTION

The **delete** statement erases the specified database entity *and all data associated with that entity*. Therefore, you must be very sure that you want to delete something before you do it.

RULES

(1) If you want to delete a database:

- For VMS, use the **delete** command.
- For UNIX systems, use the **rm** command.
- For AEGIS systems, use the **dlf** command.

(2) You can delete a field that was defined in a **define field** or **define relation** statement. However, because fields are included in a relation, you must first delete the field from each relation in which it is included. To do so, you must use the *drop-clause* option of the **modify relation** statement. The following statements delete a field from a relation and then from the database:

```
. tcs?
        modify relation states
          drop statehood;
        .sp
        delete field statehood;
```

You must explicitly delete a field defined in a **define relation** statement after dropping it from any other relations in which it is used. For example, suppose you defined the field BIRTH_DATE in an EMPLOYEES relation, subsequently included it in another relation, and then decided not to keep it. The following sequence deletes the field from all its instances and then finally from the database itself:

```
. tcs?
        modify relation employees
          drop birth_date;
        .sp
        modify relation demographics
          drop birth_date;
```

```
.sp
delete field birth_date;
```

Do not drop fields from relations unless you are sure that nothing else depends on their data. Dropping fields causes programs that depend on them to fail.

(3) You can delete any index you want, but if you delete an index that someone else is using, the other user's program will get an unrecoverable error. The following statement deletes an index:

```
. tcs?
        delete index idx4;
```

(4) You can delete any relation you want, but if you delete a relation that someone else is using, the other user's program will get an unrecoverable error. Because the **delete relation** statement removes a relation and all its records, you should use this statement with caution. The following statements delete relations:

```
. tcs?
        delete relation gudgeons;
        .sp
        delete relation non_eeoc_approved_data;
```

(5) You can delete a security class without first deleting it wherever it is referenced. The objects associated with the deleted security class are then unprotected.

(6) treats views much like relations. However, because a view is only a virtual relation, the effect of deleting a view that is being used by someone else is less catastrophic than deleting a relation that is being used. In general, when you delete a view, other users should not encounter any problems if they are already running their programs. However, if they start up a program that references the deleted view, the program fails when it tries to compile the request that mentions that view.

The following statements delete views:

```
. tcs?
        delete view population_density;
        .sp 0.5
        delete view geo_cities;
        .sp 0.5
        delete view riv_vu;
```

SEE ALSO

See Chapter 8 in this manual.

DIAGNOSTICS

See the entry for **gdef** in this manual.

NAME

field-attributes −defining field attributes

SYNTAX

field-attributes ::= *datatype* [*comments* | *edit-string* | *missing-value* | *query-name* | *security-class* | *valid-if*] ...

DESCRIPTION

The *field-attributes* clause describes the characteristics of fields defined or modified by the following statements:

- **define field** (all of the above clauses except *security-class*)
- **modify field** (all of the above clauses)
- **define relation** (all of the above clauses)
- **modify relation** (*comments* and *security-class* clauses only)

A syntactic and semantic description of each of these clauses follow.

SEE ALSO

See Chapters 5 and 6 in this manual. See also the entries in this chapter for **gdef**, **define field**, **define relation**, **define view**, **modify field**, and **modify relation**, and the discussion of the *conditional-expression* in

Syntax: datatype

datatype ::= { **short** [*scale-clause*] |
long [*scale-clause*] | **float** | **double** |
char[*n*] [*subtype*] | **varying**[*n*] [*subtype*] |
date | *blob-clause* }

scale-clause ::= **scale** [-]*n*

blob-clause ::= **blob** [*subtype*] [**segment_length** *n*]

subtype ::= **sub_type** { **text** | **blr** | **fixed** | **acl** | -*n* }

Description: datatype

The *datatype* clause specifies the datatype of a field. It is the only required field attribute.

This table lists supported datatypes by language.

Datatype	BASIC	C	COBOL	FORTRAN	Pascal	PL/I
short	word	short	s9(4) comp	I*2	integer	fixed binary(15)
long	long	long	s9(9) comp	I*4	integer32	fixed
float	single	float	comp-1	real	real	float binary(24)
double	double	double	comp-2	double_ precision	double	float binary(53)
char[n]	string	char[n]	pic x (n)	character dimension(n)	array[1...n] of char	character(n)
varying[n]	string	char[n]	pic x (n)	character dimension(n)	array[1...n] of char	character(n)
date	gds_$quad_t	gds_$quad	s9(18)	I*4 dimension(2)	gds_$quad	gds_$quad
blob	gds_$quad_t	gds_$quad	s9(18)	I*4 dimension(2)	gds_$quad	gds_$quad

This table lists the datatypes by size and range/precision.

Datatype	Size	Range/Precision
short	16 bits	-32768 to 32767
long	32 bits	-2**31 to (2**31)-1
float	32 bits	approx. 7 decimal digits
double	64 bits	approx. 15 decimal digits
char	n bytes	0 to 32767 characters
varying	varies	0 to 32767 characters
date	64 bits	1 January 100 to 11 December 5941
blob	64 bits	none

Both the date and blob datatypes are represented above by **gds_$quad**, a quantity for which allocates 64 bytes of storage. However, this quantity is functionally different for dates and blobs:

- For the date datatype, **gds_$quad** represents the date encoded in 64 bits. The GDML library includes two routines, **gds_$encode** and **gds_$decode**, for date manipulation. See the for more information.
- For the blob datatype, **gds_$quad** represents an identifier that points to the actual blob data. The format of the data depends on the application.

  The blob identifier stored in the record is a 64-bit quantity. The blob itself is of unlimited size; a blob can exceed 65,535 bytes and is limited only by the amount of physical storage available.

Varying string is a character datatype that includes a count at the beginning. This datatype is not directly supported by some host languages.

N O T E

When you use an datatype that is not supported by your host language, automatically converts such fields to equivalent types that are supported. To ensure that variables you define match the datatypes in database fields, use **based on** clause to establish the datatype of your variables. See the for more information about the **based on** clause.

Arguments: datatype

scale-clause Specifies the power of 10 by which multiplies the stored integer value for use by **qli**, COBOL, and PL/I.

For example, a negative scale of two means that there should be a decimal point two places to the left of the digits.

blob-clause Provides the characteristics of blob fields. The optional **segment_length** clause specifies a segment length that system components use for various purposes. For example, **gpre** uses this value to set up a buffer for data transfer between the calling program and the and **qli** uses the segment length to format its display.

Gdef provides a default value of 80 if you do not include the **segment_length** clause. If you update the system relations directly and leave the segment length missing, **gpre** and **qli** supply lengths of 512 and 40, respectively, for their own purposes.

subtype-clause For blob fields, there are three predefined subtypes: **text**, **blr** (request language statements), and **acl** (access control lists). **Qli** uses the subtype to determine how it should display a blob. If your application requires special blob handling, you can define your own subtype; the range of negative values from -1 to -32768 is reserved for users.

For **char** and **varying** fields, the **fixed** subtype is defined as a convenience for C programs. Text strings are passed to C as null-terminated strings unless you specify the **string** switch when you preprocess the program. If your application requires that a field contain non-ASCII binary values that may include nulls, declare the field to have the **fixed** subtype so it will *not* be truncated at the first null byte.

Example: datatype

The following statements define fields with various datatypes:

```
define field tolerance long scale -2;
        .sp
        define field text_blurb blob sub_type text
          segment_length 60;
        .sp
        define field price long
          valid if (price > 0);
        .sp
        define field manufacturer char[10]
          valid if
            (manufacturer ne "SLEAZOLA" and
            manufacturer ne "SHODTECH" and
            manufacturer ne "SCHLOKHAUS");
        .sp
        define field encrypted_key char[20]
          sub_type fixed;
        .sp
        define relation parts
```

```
item_code char[6],
item_name char[25],
manufacturer char[10],
blurb blob segment_length 60,
price long,
        .
        .
        .
```

Syntax: comments

{ *textual-commentary* **}**

Description: comments

The *comments* clause lets you store bracketed comments about the field in the database.

Arguments: comments

textual-commentary A comment can include any of these ASCII characters:

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

Example: comments

The following statements define fields with comments:

```
. tcs?
        define field standard_date date { all-purpose date field };
        .sp
        define relation parts
               item_code char[6] { alphanumeric identifier },
               item_name char[25] { abbreviated product name },
               manufacturer char[10] { aka supplier },
               blurb blob segment_length 60
                           {  this field stores the
                              descriptions of the
                              items in inventory },
               price long,
              .
              .
              .
```

Syntax: edit string

edit_string "*edit-character...*"

edit-character :== see the tables below

Description: edit string

The *edit-string* clause specifies an alphabetic, numeric, or date format for a field or computed value. Only **qli** uses edit strings.

Alphabetic and Miscellaneous Edit String Characters

Character	Meaning
A	Any alphabetic character.
X	Any alphabetic character.
B	A blank space.
'string' "string"	Print the quoted string.

Date Edit String Characters

Character	Meaning
Y (*integer*)	The year, from right to left. For 1987, *y(1)* yields *7*, *y(2)* yields *87*, and so on.
M (*integer*)	The name of the month. The integer specifies how many of the characters in the month name to print.
N (*integer*)	The numeric month. The best value for the integer is 2.
D (*integer*)	The day of the month. The best value for the integer is 2.
W (*integer*)	The name of the day of the week. The integer specifies how many of the characters in the day name to print.
B	A blank space.

Numeric Edit String Characters

Character	Meaning
9	An ordinary digit.
*	A leading asterisk (for checks).
Z	A leading digit or blank if the leading position is zero.
H	Hexadecimal representation of character.
+	Leading plus sign. Prints leading sign for positive and negative numbers.
-	Leading minus sign. Prints leading sign for negative only.
$	Leading dollar sign. Multiple dollars sign float.
(( ))	Parentheses to be printed around negative numbers.
DB	Debit.
CR	Credit.
.	Decimal point.
B	Blank space.
,	Comma for thousands, millions, etc.

N O T E

Leading signs (dollar sign, plus, and so on) take up a character space. Therefore, the number ‘‘123.45’’ with an edit string of ‘‘$$$.99’’ overflows, printing ‘‘23.45’’ on old versions of **qli** and ‘‘***’’ on newer versions.

Example: edit string

The following statements define a database and three relations, each containing several fields with edit strings:

```
. ddl_134.gdl
        define database "stuff.gdb";
        define relation budgets
        b1 long,
        b2 long edit_string "999,999",
```

```
b3 long edit_string "((999,999))",
b4 long edit_string "-ZZZ,ZZZ,ZZ9";

define relation employee_stuff
social_security char [9] edit_string "xxx-xx-xxxx",
phone_number char [10] edit_string "(xxx)Bxxx-xxxx",
salary long edit_string "HHHHHHHHHBBB'(wow!)'";

define relation family_dates
  name varying [10],
  birth date edit_string "w(3),bd(2)bm(12)by(4)",
  wedding date edit_string "d(2)bn(2)by(4)",
  awareness date edit_string "y(4)";
```

Syntax: missing value

missing_value [**is**] { *fixed-point-number* | *quoted-string* }

Description: missing value

The *missing-value* clause provides a literal string (numeric or character) that is displayed if no value is stored for that field. If you store that value in the field, marks the field as missing. The missing value must be legitimate value for the field's datatype.

Arguments: missing value

fixed-point-number A number that is displayed as the missing value. The number must not exceed the length of the field. For integer datatypes, the number cannot include a decimal point unless the field has a scale factor. For floating datatypes, the number should include a decimal point.

quoted-string A quoted literal expression that is displayed as the missing value. The string must not exceed the length of the field.

Example: missing value

The following statements define fields with explicit missing values:

```
define field price float
          missing_value is -15.75
          valid if
            (price > 0 or
            price missing);
        .sp
        define field headwater_state
          missing_value is "??";
```

Syntax: query name

query_name [**is**] *alternate-name*

Description: query name

The *query-name* clause provides an alternate field name for use in **qli**. You can reference a field by its full name or by the query name.

You may find that the longer the name, the more likely users are to mistype it. However, for reasons of internal documentation, you might want to keep the name as descriptive as possible. Therefore, you can use a query name to rename the field to something easier to type.

Arguments: query name

alternate-name A query name can contain up to 31 alphanumeric characters, dollar signs ($), and underscores (_). However, it must start with an alphabetic character.

Example: query name

The following statement defines a field with a query name:

```
define field longitude_degrees char[2]
         query_name longd;
```

The following statement defines a relation and assigns a query name to two fields:

```
define relation cities
         { largest 200 population centers }
         city,
         state,
         population,
         latitude_degrees char[2]
           query_name latd,
         latitude_minutes char[3]
           query_name latm,
         latitude_compass char[1]
           query_name latd,
         longitude_degrees char[2]
           query_name longd,
         longitude_minutes char[2]
           query_name longm,
         longitude_compass char[2]
           query_name longc;
```

Syntax: security class

security_class *class-name*

Description: security class

The *security-class* clause associates a security class with a field in a relation or a view.

You can associate a security class with a field only in a **define relation**, **modify relation**, **define view**, or **modify view** statement. You cannot use a *security-class* clause in a **define field** statement, because the security rules apply to a field in a relation, and are not a global characteristic.

Arguments: security class

class-name Names the security class you want to associate with the field. The security class must be defined in a **define security_class** statement.

Example: security class

The following statement associates a security class with a field in a relation:

```
. tcs?
        define security_class lmu
          { limited metadata update.  This class keeps everyone
            but Zaphod from assigning rights. }
            zaphod pdrwc,
            %.zaphod rw,
            %.%.gds r,
            zaphod.grd.gds p,
            view less_of_a_secret r;

        define relation r1 security_class lmu
          no_name char [10];
```

Syntax: valid if

valid_if (*boolean-expression***)**

Description: valid if

The *valid-if* clause provides a field-level integrity criterion that checks when it stores the record or updates the field in a record.

If the new value fails the test, the field assignment fails. Because the validation criteria are stored in the database, they eliminate the need for such checks in the programs that access the database.

A validation expression differs from a trigger in that a trigger has full context, can access fields in any relation, and can perform updates. A validation expression can only reference the field being validated and literal values.

Arguments: valid if

boolean-expression A valid Boolean expression. See the entry for *boolean-expression* in the or the

Example: valid if

The following statements define fields with validation expressions:

```
define field price long
          valid if (price > 0 or price missing);
         .sp
         define relation manufacturers
           manufacturer char[10]
           valid if
             (manufacturer ne "SLEAZOLA" and
             manufacturer ne "SHODTECH" and
             manufacturer ne "SHLOKHAUS" and
             manufacturer not missing),
                  .
                  .
                  .
```

NAME

modify database −modify a database

SYNTAX

modify database *quoted-filespec* [**{** *textual-commentary* **}**]
[**security_class** *class-name*]
[**drop security_class**];

DESCRIPTION

The **modify database** statement specifies the name of the database for which you want to change metadata.

The **modify database** statement must be the first statement in the source file or input to **gdef**.

ARGUMENTS

quoted-filespec A valid file specification enclosed in single (’) or double (") quotation marks. If the shell you regularly use is case-sensitive, make sure that you always reference the database file exactly as it is spelled in the **define database** statement.

The file specification can contain the full pathname to another node in the network. 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*

{ *textual commentary* **}** Stores the bracketed comments about the database in the database. The commentary can include any of the following ASCII characters:

- Uppercase alphabetic: *A—Z*
- Lowercase alphabetic: *a—z*
- Numerals: *0—9*

- Blanks, tabs, and carriage returns
- Special characters: ! @ # $ % ^ & * ( ) _ - + = ‘ ~ [ ] < > ; : ’ " \ | / ? . ,

class-name Associates a security class with the database. See the entry for **define security_class** in this chapter for more information.

EXAMPLE

The following statements each appear as the first statement in a source file used to define or modify a database:

```
modify database "/usr/igor/war_effort.gdb"
        { metadata last updated 7 November 1985 }
        security_class lmu;
       .sp
       modify database "atlas.gdb";
       .sp
       modify database "/usr/hector/dingies.gdb";
```

SEE ALSO

See Chapter 8 in this manual.

DIAGNOSTICS

See Chapter 3 for a discussion of errors and error handling.

NAME

modify field −change global field characteristics

SYNTAX

modify field *field-name field-attributes*;

DESCRIPTION

The **modify field** statement changes the characteristics of an existing field.

When you modify field characteristics, does not change the characteristics on disk immediately. Instead, it changes the characteristics when someone updates the field. Meanwhile, ‘‘filters’’ unchanged values so that they reflect the updated characteristics until they are actually changed.

If you want to change the characteristics of a field defined inside a relation, you may not be able to change it within that relation. For example, physical attributes, such as the datatype, can be changed only with the **modify field** statement. This restriction is due to the way **gdef** deals with fields defined in relation. In short, fields defined in relations (*locally defined fields*) are automatically added to the list of fields in the database (*globally defined fields*). Although you may define a field inside a relation, **gdef** disregards the source of the definition and makes it a globally defined field as if it were defined with a **define field** statement.

Therefore, if you want to change anything other than the field’s position, commentary as a part of that relation, query name, or security class as a part of that relation, you must do so with the **modify field** statement. **Gdef** then makes the change for all uses of that global field in all relations of the database.

Finally, if you change the datatype or length of fields upon which a **computed by** field is based, the length of the computed field does not change automatically. This does not cause a problem if you decrease the size of the base fields, but will cause a data conversion error if you increase them, particularly if the computation consists of concatenated strings. The preferred way to fix this problem is to delete and re-create the computed field. The ‘‘quick and dirty’’ way is to manually increase the value of RDB$FIELD_LENGTH of the RDB%FIELDS record for the computed field.

ARGUMENTS

field-name Names the field you want to create or the field you want to modify.

field-attributes Specifies the datatype, length or scale if appropriate, and several optional field characteristics. You can change the textual description of the field or its query name. For detailed information about the syntax and semantics of these options, see the entry for *field-attributes* in this chapter.

EXAMPLE

The following statements modify fields:

```
. tcs?
       modify field zip char[9];
```

```
.sp
modify field headwater_state
  missing_value is "??";
.sp
modify field price longword;
```

SEE ALSO

See Chapter 8 in this manual. See also the entries in this chapter for:

- *field-attributes*
- **modify relation**

DIAGNOSTICS

See Chapter 3 for a discussion of errors and error handling.

NAME

modify relation −modify a relation

SYNTAX

modify relation *relation-name* [**{** *textual-commentary* **}**] [*operation-commalist*];

operation ::= { **add field** *field-name* [*field-attributes* |
drop field *field-name* |
modify field *field-name* [*field-attributes* |
drop security_class [*security-class-name*] |
security_class *security-class-name* }

DESCRIPTION

The **modify relation** statement can change a relation's comment field, complement of fields, and local field characteristics.

ARGUMENTS

relation-name Identifies the relation you want to modify.

{ *textual-commentary* **}** Stores comments about the relation in the database. The *textual-commentary* can include any of these ASCII characters:

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

add field *field-name* Adds a field to the relation:

- If the field has already been defined in the database, *field-attributes* can specify the position of the field, textual description of the field, a query name for use with **qli**, and a security class.
- If the field does not exist elsewhere, you must specify a datatype. You can also specify an explicit missing value, a validation expression, the position of the field, textual description of the field, a query name for use with **qli**, and a security class.

The addition of fields to a relation is almost identical to the inclusion of fields when you define the relation. See the entry for **define relation** in this chapter for more information.

drop field *field-name* Removes the named field from the relation. When you delete a field from a relation, other users should not encounter any problems if they are already running their programs. However, if they

start up a program that references the deleted field, the program fails when it tries to compile the request that mentions that field.

You cannot delete fields that are used in views based on this relation without first deleting the field from those views.

modify field *field-name* Identifies the field whose relation-specific characteristics you want to change. You can change only the position of the field, the textual description, the query name, the security class of the field, and the field on which the local field is based (for fields defined with the **based on** option).

You cannot change the datatype, the missing value, or the validation expression. If you want to change one of these attributes, you must do so with the **modify field** statement.

drop security_class [*security-class-name*] Removes the named security class from the relation. If you do not specify *security-class*, **gdef** removes any security class associated with the relation.

security_class *security-class-name* Associates the specified security class with the relation.

EXAMPLE

The following example modifies a relation by adding fields, dropping fields, and modifying fields:

```
. tcs?
        modify database "test_atlas.gdb";
        .sp
        modify relation cities
          add field year_incorporated char[4]
            query_name inc
            position 6,
          add field type_of_government char[1]
            query_name gov
            valid_if
              (type_of_government = "C" or
               type_of_government = "M" or
               type_of_government missing),
          drop field population;

        modify relation states
          security_class cabinet_level;
```

SEE ALSO

See Chapter 8 in this manual. See also the entries in this chapter for:

- *field-attributes*

- **define relation**
- **modify field**

DIAGNOSTICS

See the entries for **gdef** in this chapter.

NAME

modify trigger −modify integrity check

SYNTAX

modify trigger for *relation-name*
[**store:** *trigger-action*]
[**modify:** *trigger-action*]
[**erase:** *trigger-action*]

DESCRIPTION

The **modify trigger** statement changes the action that performs automatically whenever you execute a store, modify, or erase operation on the relation.

ARGUMENTS

{ *textual-commentary* **}** Stores or modifies comments about the trigger in the database. The *textual-commentary* can include any of these ASCII characters:

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

store:
modify:
erase: Changes the trigger action performed on a store (or insert), modify (or update), or erase (or delete) operation. Each of these operations can have a separate trigger action.

trigger-action Specifies a GDML statement that executes whenever you store a new record into the relation, modify a field from a record in the relation, or erase a record from the relation. See the for information about GDML data manipulation.

EXAMPLES

The following statements modify the erase trigger associated with a relation:

```
. tcs?
        modify database "not_yachts.gdb";
        .sp
        modify trigger for widgets
          erase:
```

```
            store x in log
                x.what = "GONZO";
                x.name = old.name;
                x.old_number = old.number;
                x.when = "today";
            end_store;
        end_trigger;
```

SEE ALSO

See Chapter 8 in this manual.

DIAGNOSTICS

See Chapter 3 for a discussion of errors and error handling.

NAME

modify view −modify view

SYNTAX

modify view *view-name* [**{** *textual-commentary* **}**] [*operation-commalist*]

operation ::= { **drop field** *field-name* |
drop security_class [*security-class-name*] |
security_class *security-class-name* }

DESCRIPTION

The **modify view** statement:

- Drops a field from a view
- Drops a security class for a view
- Adds a security class for the view

ARGUMENTS

view-name Identifies the view you want to change.

{ *textual-commentary* **}** Stores the bracketed comments about the view in the database. The *textual-commentary* can include any of the following ASCII characters:

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

drop field *field-name* Removes the named field from the view, but not from the source relation(s). You cannot delete fields that are used in views based on this view without first deleting the field from those views.

drop security_class [*security-class-name*] Removes the named security class. If you do not specify *security-class*, **gdef** removes any security class associated with the view.

security_class *security-class-name* Associates the specified security class with the view.

EXAMPLES

The following statement removes a field, drops a security class, and adds a new security class:

```
. tcs?
        modify view geo_cities
          { new comment goes here }
          drop field altitude,
          drop security_class,
          add security_class top_secret;
```

SEE ALSO

See Chapter 8 in this manual. See also the entries in this chapter for:

- *field-attributes*
- **define view**
- **define security_class**

DIAGNOSTICS

See Chapter 3 for a discussion of errors and error handling.