InterBase
Data Definition Guide

Disclaimer

Borland International, Inc. (henceforth, Borland) reserves the right to make changes in specifications and other information contained in this publication without prior notice. The reader should, in all cases, consult Borland to determine whether or not any such changes have been made.

The terms and conditions governing the licensing of InterBase software consist solely of those set forth in the written contracts between Borland and its customers. No representation or other affirmation of fact contained in this publication including, but not limited to, statements regarding capacity, response-time performance, suitability for use, or performance of products described herein shall be deemed to be a warranty by Borland for any purpose, or give rise to any liability by Borland whatsoever.

In no event shall Borland be liable for any incidental, indirect, special, or consequential damages whatsoever (including but not limited to lost profits) arising out of or relating to this publication or the information contained in it, even if Borland has been advised, knew, or should have known of the possibility of such damages.

The software programs described in this document are confidential information and proprietary products of Borland.

Restricted Rights Legend. Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subdivision (b) (3) (ii) of the Rights in Technical Data and Computer Software clause at 52.227-7013.

Corporate Headquarters: Borland International Inc., 100 Borland Way, P. O. Box 660001, Scotts Valley, CA 95067-0001, (408) 438-5300. Offices in: Australia, Belgium, Canada, Denmark, France, Germany, Hong Kong, Italy, Japan, Korea, Latin America, Malaysia, Netherlands, New Zealand, Singapore, Spain, Sweden, Taiwan, and United Kingdom.

Software Version: V3.0

Current Printing: October 1993
Documentation Version: v3.0.1

Reprint note

This documentation is a reprint of InterBase V3.0 documentation. It contains most of the information from *InterBase Previous Versions Documentation Corrections* and *InterBase Version* 3.2 *Documentation Corrections* and a new index. For information on features added since InterBase Version V3.0, consult the appropriate release notes.

Table Of Contents

Preface

This manual describes how to define and modify InterBase databases.

Who Should Read this Book

You should read this book if you want to learn how to define or modify InterBase databases. Before you read this book, you should have previous experience with programming languages and should understand the concepts of the relational data model.

Using this Book

This book contains the following chapters and appendixes:

Chapter 1	Provides a brief overview of data definition with InterBase.
Chapter 2	Describes how to design your database.
Chapter 3	Describes how to create an InterBase database.
Chapter 4	Describes how to define fields and describes how and when to the use various field attributes.
Chapter 5	Describes how to define fields for relations and how to modify and delete relations.
Chapter 6	Describes how to define, modify, and delete views and indexes. It also discusses how to add indexes to existing relations.
Chapter 7	Describes database integrity rules and the InterBase mechanisms for enforcing those rules.
Chapter 8	Describes how to secure data and metadata by using InterBase security classes.
Chatper 9	Describes how to code, define, and access user-defined functions, and how to create a function library for storing the functions.
Chapter 10	Introduces the InterBase event alerter mechanism, describes how this mechanism works, and discusses event transaction control.
Chapter 11	Describes how to modify metadata with dynamic DDL.
Chapter 12	Describes how to use program interfaces other than **gdef** to define data.
Appendix A	Presents an overview of the InterBase system relations and presents a detailed description of each relation.
Appendix B	Shows a sample database definition.

Text Conventions

This book uses the following text conventions.

boldface

Indicates a command, option, statement, or utility. For example:

- Use the **commit** command to save your changes.
- Use the **sort** option to specify record return order.
- The **case_menu** statement displays a menu in the forms window.
- Use **gdef** to extract a data definition.

italic

Indicates chapter and manual titles; identifies filenames and pathnames. Also used for emphasis, or to introduce new terms. For example:

- See the introduction to SQL in the *Programmer's* Guide.
- /usr/interbase/lock_header
- Subscripts in RSE references *must* be closed by parentheses and separated by commas.
- C permits only *zero-based* array subscript references.

`fixed width font`

Indicates user-supplied values and example code:

- $run sys$system:iscinstall
- add field population_1950 long

UPPER CASE

Indicates relation names and field names:

- Secure the RDB$SECURITY_CLASSES system relation.
- Define a missing value of *X* for the LATITUDE_COMPASS field.

Syntax Conventions

This book uses the following syntax conventions.

{braces}	Indicates an alternative item: • option::= {vertical | horizontal | transparent}
[brackets]	Indicates an optional item: • dbfield-expression[not]missing
`fixed width font`	Indicates user-supplied values and example code: • `$run sys$system:iscinstall` • add field population_1950 long
commalist	Indicates that the preceding word can be repeated to create an expression of one or more words, with each word pair separated by one comma and one or more spaces. For example, field_def-commalist resolves to: field_def[,field_def[,field_def]...]
italics	Indicates a syntax variable: **create_blob** *blob-variable* **in** *dbfield-expression*
|	Separates items in a list of choices.
⇓	Indicates that parts of a program or statement have been omitted.

InterBase Documentation

The InterBase Version 3.0 documentation set contains the following books:

Getting Started with InterBase (INT0032WW2179A) provides an overview of InterBase components and interfaces.

Database Operations (INT0032WW2178D) describes how to use InterBase utilities to maintain databases.

Data Definition Guide (INT0032WW2178F) describes how to create and modify InterBase databases.

DDL Reference (INT0032WW2178E) describes the function and syntax for each of the data definition language clauses and statements. It also lists the standard error messages for **gdef**.

DSQL Programmer's Guide (INT0032WW2179C) describes how to program with DSQL, a capability for accepting or generating SQL statements at runtime.

Forms Guide (INT0032WW2178A) describes how to create forms using the InterBase forms editor, **fred,** and how to use forms in **qli** and GDML applications.

Programmer's Guide (INT0032WW2178I) describes how to program with GDML, a relational data manipulation language, and SQL, an industry standard language.

Programmer's Reference (INT0032WW2178H) describes the function and syntax for each of the GDML and InterBase supported SQL clauses and statements. It also lists the standard error messages for **gpre**.

Qli Guide (INT0032WW2178C) describes the use of **qli**, the InterBase query language interpreter that allows you to read to and write from the database using interactive GDML or SQL statements.

Qli Reference (INT0032WW2178B) describes the function and syntax for each of the data definition, GDML, and SQL clauses and statements that you can use in **qli.**

Sample Programs (INT0032WW2178G) contains sample programs that show the use of InterBase features.

Master Index (INT0032WW2179B) contains index entries for the entire InterBase Version 3.0 documentation set.

In addition, platform-specific installation instructions are available for all supported platforms.

Chapter 1
Introduction

This chapter introduces data definition with InterBase.

Overview

InterBase provides a number of tools for defining data:

- **Gdef**, the data definition compiler, gives you complete data definition capabilities. You can use **gdef** to create:
  - The database itself
  - Global or local fields
  - Relations
  - Views that consist of fields from one or more relations
  - Indexes on relations and views
  - Security for databases, relations, views, and fields
  - Triggers to specify actions that InterBase performs automatically when you store, modify, or erase a record

- **Qli**, the InterBase query language, provides a subset of **gdef**'s capabilities. You can use **qli** to define:
  - The database itself
  - Global and local fields
  - Relations
  - Views that consist of fields from one or more relations
  - Indexes on relations and views
  - Security for relations
- Embedded SQL and Dynamic SQL (DSQL) also provide a subset of **gdef**'s capabilities. You can use embedded SQL statements to define:
  - The database itself
  - Local fields
  - Relations
  - Views that consist of fields from one or more relations
  - Indexes on relations and views
  - Security for relations

  You can use DSQL statements to define all of the objects listed above, except for a database.
- Dynamic DDL (DYN) gives you the ability to embed statements generated by **gdef** into host-language programs. This lets you extend database definitions to support new features and then send users a program that updates their databases in place.

No matter which data definition tool you use, InterBase uses *system relations* to hold the data definitions, or *metadata*. When you use **gdef**, **qli**, and dynamic DDL to define and change metadata, the definitions and changes are automatically reflected in the database at commit time. When you use SQL or DSQL to define and change metadata, the definitions and changes are automatically reflected in the database after successful compilation. Current metadata is available to every program. This gives you an *integrated active data dictionary* of your metadata.

System relations are structured exactly like user relations, so you can use the program interfaces **qli**, SQL, and GDML to perform the same operations on both data and data definitions.

The Advantages of Using Gdef

Using **gdef** and its data definition statements is a quick, reliable way to define or modify metadata. **Gdef**'s data definition statements resemble data declarations from programming languages such as C and Pascal. First, you create a file, called the *source file*, that contains DDL statements. Then, you compile the source file with **gdef**. If **gdef** compiles the source file without errors, it creates a database.

If you want to change an existing database in any way, such as adding a field to a relation or dropping a view, you can create a source file that contains **gdef**'s metadata modification statements and then compile it. You can also use **gdef** interactively to make ad-hoc changes.

Whether you choose to use **gdef** or another metadata interface, you need to know what metadata features and capabilities are available in InterBase. Therefore, you should read this manual for a discussion of creating databases and defining fields, relations, views, indexes, triggers, access control, user-defined functions, and event alerters. Although all examples are in the form of **gdef**'s DDL statements, these chapters help you write programs that deal with metadata. Chapter 12, *Using Other Interfaces to Define Data* presents examples of using the program interfaces to define metadata.

Minimal Database Definition

At a minimum, a database consists of relations and fields. For example, the following DDL statements define a usable database, *phones.gdb*, that consists of a single relation (PHONE_NUMBERS) with four fields:

```
define database "phones.gdb";

define relation phone_numbers
     first_name char[10],
     last_name char[20],
     area_code char[3],
     phone char[8];
```

These statements can be incorporated in a file, such as *phone.gdl*, and passed to **gdef**. For example:

```
% gdef phone.gdl
```

The statements can also be typed interactively after you invoke **gdef**. For example:

```
% gdef
GDEF> define database 'phones.gdb';
GDEF> define relation phone_numbers
CON>   first_name char[10],
CON>   last_name  char[20],
CON>   area_code  char[3],
CON>   phone char[8];
CON>  <EOF>
GDEF>
```

Use your system's end-of-file character or type "exit" to terminate the definition.

In both this interactive and the preceding case, once **gdef** processes these data definitions, you can store records in the PHONE_NUMBERS relation. For example:

```
QLI> ready phones.gdb
QLI> store phone_numbers
Enter FIRST_NAME: Ivan
Enter LAST_NAME: Pillow
Enter AREA_CODE: 617
Enter PHONE: 555-6600
QLI>
```

As your application grows and more people start using it, you will probably add more relations to store related data, views to simplify data access, security classes to limit access, and indexes to improve performance. You can add these as needed.

For More Information

For more information on data definition concepts and data definition through **gdef** and dynamic DDL, continue reading this manual. If you need to look up the syntax of a particular DDL statement, refer to the *DDL Reference*.

For more information on other data definition tools provided by InterBase, refer to the *Qli Guide* and the *Programmer's Guide*.

Chapter 2
Designing a Database

This chapter discusses some aspects of logical database design.

Overview

The basic principle of logical database design is to break large heterogenous groupings of data into smaller pieces, so that each *relation* deals with closely *related* data.

The suggestions in this chapter are really guidelines. You *must* remove repeating items and groups, if you want to follow the relational model. The other steps depend on your application needs.

The design information presented in this chapter is based on the relational data model.

Analyzing System Requirements

The first step in designing a database is to study your company's data requirements. A basic systems analysis usually results in an understanding of:

- What the data items are
- What each data item is used for
- How each data item relates to the others

Once you have this information, you can begin to consider other factors, such as the form of relations and the data model used by InterBase.

There are many methodologies you can use to conduct a systems analysis. For information on these methodologies, refer to the many books on this subject.

Assigning a Primary Key

The second step in designing a database is to establish a *primary key* for each grouping of data. A primary key is a data item that uniquely identifies a group of related data. Sometimes you need more than one data item to uniquely identify a group of related data. In this case, you would use a *concatenated key* as your primary key.

For example, consider an order system developed for a mail order business. The predecessor to the InterBase version of this system relied on a very large flat file called CUST_ORDER. This file contained a single record for each order. Each record contained order information, customer data, and payment data. Each record also contained repeating fields (ITEM_NUMBER, ITEM_PRICE, ITEM_NAME, and ITEM_QUANTITY) that described the items in the order:

```
CUST_ORDER record
   ORDER_NUMBER
   CUSTOMER_NUMBER
   ENTRY_DATE
   CUSTOMER_LAST_NAME
   CUSTOMER_FIRST_NAME
   STREET_ADDRESS
   CITY
   STATE
   ZIP
   PHONE
   ITEM_NUMBER
   ITEM_PRICE
   ITEM_NAME
   ITEM_QUANTITY
   AMOUNT_OF_ORDER
   METHOD_OF_PAYMENT
```

For this grouping of data, you might make ORDER_NUMBER the primary key. It's a good idea to mark the data item or items that make up the primary key:

```
CUST_ORDER relation
   *ORDER_NUMBER
    CUSTOMER_NUMBER
    ENTRY_DATE
    CUSTOMER_LAST_NAME
    CUSTOMER_FIRST_NAME
    STREET_ADDRESS
    CITY
    STATE
    ZIP
```

```
PHONE
ITEM_NUMBER
ITEM_PRICE
ITEM_NAME
ITEM_QUANTITY
AMOUNT_OF_ORDER
METHOD_OF_PAYMENT
```

Later, when you define the database, you should define a unique index for each primary key. For more information on defining a unique index, refer to Chapter 6, *Defining Views and Indexes*.

Normalizing Your Data

Once you have analyzed your system and assigned primary keys to your data groupings, you can begin to *normalize* your data. This process reduces data to its simplest form and helps to prevent data anomalies, which can compromise the integrity of your database.

Through normalization, you develop relations that consist of:

- A primary key
- A set of data items whose values are determined solely by the value of the primary key

Follow these steps to normalize your data:

- Eliminate repeating groups
- Remove partial-key dependencies
- Remove transitive dependencies

Eliminating Repeating Groups

The first step in the normalization process is to eliminate repeating groups. A *repeating group* is a group of data items that contain more than one value for each occurrence of the primary key.

For example, in the CUST_ORDER relation shown above, the data items that describe an ordered item form a repeating group:

```
ITEM_NUMBER
ITEM_PRICE
ITEM_NAME
ITEM_QUANTITY
```

These items form a repeating group, because for each individual ORDER_NUMBER, there may be many items ordered.

You eliminate repeating groups by moving the group into a new relation. You must assign a primary key to the new relation. Usually, the primary key is a concatenated key that includes the primary key of the source relation (in this case, CUST_ORDER) and one or more data items in the target relation.

The example below shows how to eliminate the repeating group in the CUST_ORDER relation:

```
CUST_ORDER relation
   *ORDER_NUMBER
```

```
    CUSTOMER_NUMBER
    ENTRY_DATE
    CUSTOMER_LAST_NAME
    CUSTOMER_FIRST_NAME
    STREET_ADDRESS
    CITY
    STATE
    ZIP
    PHONE
    AMOUNT_OF_ORDER
    METHOD_OF_PAYMENT

ORDER_ITEM relation
   *ORDER_NUMBER
   *ITEM_NUMBER
    ITEM_PRICE
    ITEM_NAME
    ITEM_QUANTITY
```

Removing Partially Dependent Fields

The second step in the normalization process is to remove any fields that are logically dependent on only part of a concatenated key. Such fields have a *partial-key dependency* on the concatenated key field.

For example, the ORDER_ITEM relation above has a concatenated key composed of ORDER_NUMBER and ITEM_NUMBER. Some fields in ORDER_ITEM are not dependent on the combination of those fields. They are dependent only on the ITEM_NUMBER field. Once again, this division saves effort and space because item information is stored in one place, rather than with every order.

The example below shows how to split item information from order information in the ORDER_ITEM relation:

```
CUST_ORDER relation
   *ORDER_NUMBER
    CUSTOMER_NUMBER
    ENTRY_DATE
    CUSTOMER_LAST_NAME
    CUSTOMER_FIRST_NAME
    STREET_ADDRESS
    CITY
    STATE
    ZIP
```

```
    PHONE
    AMOUNT_OF_ORDER
    METHOD_OF_PAYMENT

ORDER_ITEM relation
   *ORDER_NUMBER
   *ITEM_NUMBER
    ITEM_QUANTITY

CATALOG_ITEM relation
   *ITEM_NUMBER
    ITEM_NAME
    ITEM_PRICE
```

Removing Transitively Dependent Fields

The third step in the normalization process is to remove any fields that are not logically related to the key field. Such fields are said to have a *transitive dependency* on the key field.

For example, many fields in the CUST_ORDER relation are not related to the ORDER_NUMBER field. Instead, these fields are related to CUSTOMER_NUMBER.

In addition to being tidier and fitting with normalization theory, separating the customer information from the order information saves you typing time and storage space for each new order.

The example below shows how to split customer information from the CUST_ORDER relation:

```
ORDER relation
   *ORDER_NUMBER
    CUSTOMER_NUMBER
    ENTRY_DATE
    AMOUNT_OF_ORDER
    METHOD_OF_PAYMENT

CUSTOMER relation
    *CUSTOMER_NUMBER
     CUSTOMER_LAST_NAME
     CUSTOMER_FIRST_NAME
     STREET_ADDRESS
     CITY
     STATE
```

```
      ZIP
      PHONE

ORDER_ITEM relation
   *ORDER_NUMBER
   *ITEM_NUMBER
    ITEM_QUANTITY

CATALOG_ITEM relation
   *ITEM_NUMBER
    ITEM_NAME
    ITEM_PRICE
```

As you can see from the final versions of the relations, fields that uniquely identify one relation can also appear in other relations. In the language of relational database management, a unique identifier such as ORDER_NUMBER is a *primary key* in the ORDER relation, but a *foreign key* in ORDER_ITEM. Primary and foreign keys tie a relational database together.

For example, the occurrence of the ORDER_NUMBER and CUSTOMER_NUMBER fields in the ORDER relation, and the CUSTOMER_ NUMBER field in the CUSTOMER relation lets you reconstitute the previous version of the CUST_ORDER record from the new relations. You can do this by using the relational *join* operation. For more information on the join operation, refer to Chapter 6, *Defining Views and Indexes*.

Developing a Prototype Design

You should always plan to do numerous versions of your database design. Because your goal is to produce a system that meets users' needs, you need to work with the system's eventual users on each step of the database design process.

One strength of a database management system is that data independence preserves the good parts of a prototype while you deal with unforeseen circumstances. A relational database management system is particularly adept at letting a database design evolve.

For More Information

For more information about systems analysis and relational database design, see one of the books available on these subjects.

Chapter 3
Creating a Database

This chapter discusses how to create an InterBase database. First, it provides an overview of the networks for which InterBase is designed. Next, it tells you how to create database files, how to define an InterBase database, how to extract metadata, and how to handle **gdef** errors.

Overview

InterBase databases are designed for sharing data across networks and for multiple database access. These topics are discussed on the following page.

Sharing Data Across Networks

InterBase supports data sharing on both:

- *Homogeneous* networks, which consist of a single type of hardware. For example, a homogeneous network may consist of DEC VAX or SunOS 3.5 workstations, but not both.
- *Heterogeneous* networks, which consist of many types of hardware. Such a network might include DEC, Sun, Apollo, and HP machines. These machines must be connected either by TCP/IP or DECnet (DEC machines only).

This means you can keep your databases wherever they are needed in the network, but still give users on other machines in the network complete access to data. For example, you can store a design database on one machine, a specifications database on another, and an engineering personnel database on yet another. You can read and write to each of those databases from any node within the network.

Accessing Multiple Databases

Each InterBase data manipulation interface supports multiple-database access within a single transaction. To guarantee the consistency of the databases during a multiple database transaction updating data, InterBase supports a *two-phase commit* on the transaction. This involves:

1. Checking each participating node to see if anything stands in the way of committing the transaction.
2. Making the database changes permanent when all partipating nodes check in.

The net result of these capabilities is that you can develop distributed applications that share data among the machines you happen to be using.

For more information on InterBase transaction control, refer to the chapter on InterBase transaction management in the *Programmer's Guide.*

Creating Database Files

You can store an InterBase database in a single file or in multiple files:

- **Gdef** creates *single-file* databases by default. Most databases, at least in their early stages of development and prototyping, don't exceed the storage capacity of a single disk drive.
- **Gdef** also lets you specify a *multiple-file* database. You can specify a multiple-file database when you first create the database, or when the database grows too large for a single drive, A multiple file database can have its files distributed over several disks.

In both cases, the database file or files contain both user data and the integrated data dictionary. Instructions for creating single- and multiple-file databases are presented below, followed by instructions for overriding the default database page size.

Creating a Single-File Database

The following statement directs **gdef** to create a single-file database with the specified filename:

```
GDEF> define database "atlas.gdb";
```

You may want to specify a symbolic link (UNIX systems) or a logical name (VMS systems) rather than a filename, to gain added flexibility in database administration.

For example, the following statements specify a symbolic link for UNIX sytems and define a database using that link:

```
% ln -s databases/atlas.gdb sample_db
GDEF> define database sample_db;
```

The following statement defines the database identified by the VMS logical name *atlas$database*:

```
GDEF> define database "atlas$database";
```

Creating a Database Remotely

You can create a database remotely in a network connected by DECnet, TCP/IP, or an Apollo ring. For example, suppose you are on an Ultrix node and want to create a database on a VMS node called HECTOR. The nodes are connected with DECnet. The following commands begin the creation of a database on HECTOR:

```
csh# gdef
GDEF> define database "hector::dua1:[user.data]atlas.gdb";
```

The vertical arrow ( ) represents other data definition statements that define relations, fields, indexes, views, triggers, and access control. These subjects are discussed later in this manual.

The form of the pathname to the remote node varies by operating system and type of network:

- Between UNIX nodes connected by TCP/IP, the node name is followed by one colon (:) and then the file specification on the remote node. For example:

  ```
  "gustav:/usr/kathy/test/atlas.gdb"
  ```

- Between DECnet nodes, the node name is followed by two colons (::) and then the file specification on the remote node. For example:

  ```
  "hector::dua1:[user.data]atlas.gdb"
  "gustav::/usr/kathy/test/atlas.gdb"
  ```

- Between VMS and other nodes connected by TCP/IP, the node name is followed by a caret (^) and then the file specification on the remote node. For example:

  ```
  "apollo^/dev_1/kathy/test/atlas.gdb"
  ```

- Between Apollo nodes on the same token ring, the node name is preceded by two slashes (//) and then followed by the file specification on the remote node. For example:

  ```
  "//pooch/dev_1/kathy/test/atlas.gdb"
  ```

- Between Apollo nodes connected by TCP/IP, the node name is followed by a colon (:) and then the file specification on the remote node. For example:

  ```
  "pooch:/dev_1/kathy/text/atlas.gdb"
  ```

Tip for NFS Users

It's best to create a database on the node where you want the database to reside. With NFS, file structures on remote nodes appear to be local.

When you create a database, the database format matches the node where the file physically resides. All access to the database takes place through that node, rather than through nodes that have the database file mounted through NFS.

Creating a Multiple-File Database

A multiple-file database consists of a *primary file* and one or more *secondary files*. InterBase uses the primary file as the first database file. When the pages on the primary file fill up, InterBase allocates a secondary file. Then when the pages on the secondary file fill up, InterBase allocates another secondary file. InterBase continues this process until the file allocations run out.

You can use secondary files only for overflow purposes. You can't specify what information goes into each file, because InterBase handles this automatically.

You can either specify secondary files when the database is defined, or add them as they become necessary.

Specifying Page Ranges and File Lengths

When you define a secondary file, you must declare a range of pages to be stored in that file. You can do that by specifying either:

- A length in pages for each file
- A starting page number for each secondary file
- A combination of length and starting page numbers

For example, the following statement creates a database that's stored in four 10,000-page long files:

```
define database "world_atlas.gdba"
  file "world_atlas.gdbb" starting at page 10001
  length 10000 pages
  file "world_atlas.gdbc"
  length 10000 pages
  file "world_atlas.gdbd";
```

You should be aware of the following considerations when you specify a file length:

- If you don't declare a length for a file, you must declare a starting position for the next file. In the preceding example, the primary file specification does not provide a length. However, the first secondary file lists a starting position.
- If you declare a length that's inconsistent with the starting page number, **gdef** chooses the value that makes the first file longer.

  For example, suppose the preceding example took this form, in which the primary file is 10,000 pages long, but the first secondary file starts at page 5000:

```
define database "world_atlas.gdba" length 10000
   file "world_atlas.gdbb" starting at page 5000
      length 10000 pages
   file "world_atlas.gdbc"
      length 10000 pages
   file "world_atlas.gdbd";
```

  Gdef makes the primary file 10,000 pages long and starts the first secondary file at page 10,001.
- If you choose to describe page ranges in terms of length, you must list the files in the order in which they should be filled. In the example above, the files are filled in the order *world_atlas.gdba*, *world_atlas.gdbb*, *world_atlas.gdbc*, and *world_atlas.gdbd*.

Specifying Additional Files

InterBase extends a multiple file database as follows:

- It checks that the page it's about to create is in the page range defined for the file to which it's adding. If not, it opens the next file.
- If a database file is full and no other file is specified, InterBase extends the last file beyond the limit you specify, until the disk space runs out:
  - — On Apollo and UNIX systems, InterBase extends the file one page at a time.
  - — On VMS sytems, InterBase first doubles the file size up to 16,000 bytes. Then it extends the file 16,000 bytes at a time.

Therefore, to avoid overfilling the device that holds the last database file, be sure you specify enough secondary files. You specify these files by using the **modify database** statement, as described in the *DDL Reference*.

In most cases, you will probably want to put secondary files on separate disks. However, you must ensure that all files in a database can be accessed directly by one program:

- On Apollo systems, all database files must be in the same ring.
- On UNIX systems all database files must be in the same file system, and can't include disks that are NFS mounts.
- On VMS systems, all database files must be on disks mounted by the same host or cluster-wide devices available to that host.

Overriding the Default Page Size

When you create a database, you can override the default page size of 1024 bytes. You can specify a page size of 1024, 2048, 4096, or 8192 bytes. InterBase rounds other page sizes up to the nearest increment with a warning. It returns an error for sizes above 8192.

Consider using a larger page size when:

- You have an indexed relation that has a large number of record occurrences. The advantage of a large page size is that it allows a more shallow tree structure in the index. Each index bucket is one page long, so longer pages mean larger buckets and fewer levels in the hierarchy.

  For example, if you have more than 50,000 records in an indexed relation, you should try a page size of 2048 bytes, rather than the default of 1024 bytes.
- You have many large records that contain non-repetitive data. InterBase performs better if each record fits on a page. Large records that contain non-repetitive data do not compress as compactly as others. Thus, they take up more space, and may not fit on a single 1024 byte page.
- You have large blob fields. Blob storage and retrieval is most efficient when the entire blob fits on a single page. If an application typically stores blobs between 1K and 2K, a page size of 2048 bytes is preferable to one of 1024 bytes.

To change a page size on an existing database, you must use **gbak**, rather than **gdef**. For more information on changing a page size, see the reference chapter in *Database Operations*.

Creating a Shadow File

A *shadow file* is a physical copy of a database that resides on a disk. Once enabled, the shadow file maintains a duplicate copy of the database. This copy is always in sync with the database. You use a shadow file to recover from hardware failures.

You create a shadow file by using the **define shadow** command in **gdef**. You can define either a single shadow file or a set of shadow files:

- To define a single shadow file, specify a shadow set number and a pathname for the shadow file. For example:

```
define shadow 1 auto 'atlas_shadow';
```

- To define a shadow set, specify a shadow set number, a pathname for the shadow files, and either a file length or a starting page number for each of the secondary files. For example, to define a shadow set consisting of 3 files, type:

```
define shadow 1 auto 'atlas.shadow' starting at page 0
   file 'atlas.shadow1'starting at page 100
   file 'atlas.shadow2' starting at page 200;
```

  As an alternative, type:

```
define shadow 2 auto 'atlas.shadow' length 100
   file 'atlas.shadow1' length 100
   file 'atlas.shadow2' length 100;
```

For more information on shadow files, refer to the chapter on disk shadowing in the *Database Operations* guide.

Defining a Database

You define a database by compiling data definition (DDL) statements with **gdef**. When you define a database, InterBase creates a file for the database and populates it with *system relations*. These relations describe the structures of both internal data and user data.

The first statement you pass to **gdef** determines whether it creates or modifies a database.

- If the first statement passed is **define database**, **gdef** creates a file with the name specified in this statement. InterBase populates that file with system relations as determined by the DDL statements that follow the **define database** statement.

  In the case of multiple-file databases, **gdef** creates files with the names and characteristics specified in the **define database** statement for each secondary file.
- If the first statement passed is **modify database**, **gdef** modifies the specified database file as determined by the DDL statements that follow the **modify database** statement.

Supported DDL operations are listed below, followed by a discussion of **gdef** syntax and instructions for entering DDL Statements.

Supported DDL Operations

You can use **gdef** to:

- Define a database, field, relation, view, index, trigger, user-defined function, blob filter, and shadow file
- Modify a database, field, relation, view, index, trigger, user-defined function, blob filter, and shadow file definition
- Delete a database, field, relation, view, index, trigger, user-defined function, blob filter, and shadow file definition
- Add a field to an existing relation
- Secure your data

A subset of **gdef** syntax follows. This syntax is described in detail in the *DDL Reference*.

Operating System	Syntax
Apollo AEGIS	gdef [-r] [-z] [-p *integer*] [*filespec*]
UNIX	gdef [-r] [-z] [-p integer] [*filespec*]
VMS	gdef [/replace] [/z] [/page_size *integer*] [*filespec*]

Entering DDL Statements

You can either include DDL statements in a source file or input them interactively:

- To input DDL statements from a source file, you must provide its file specification. When you do this:
  - — Gdef looks for the specified file.
  - — If it can't find the file, gdef looks for the specified file with an extension of *gdl*.
  - — If it can't find that file, gdef fails.
- If you type **gdef** followed by a carriage return, **gdef** accepts input directly from your keyboard. Terminate input with the standard end-of-file character, the **quit** statement, or the **exit** statement.

The rules and conventions you use to enter DDL statements are described below, followed by instructions for entering DDL statements from a file and entering DDL statements interactively.

DDL Input Rules and Conventions

You should be aware of the following rules when you enter DDL statements:

- Punctuation in the source file or interactive input is limited to:
  - — Commas (,) to delimit items in a list
  - — Semicolons (;) to terminate statements
  - — Double and single quotes (" and ') for file specifications and literal values
- You can either define fields before you include them in relations, or you can define them when you define a relation. If you want to reference a field in a **define relation** statement, that field must precede any other **define relation** statement that references the field.
- You must define a relation before you define indexes for it.
- You must define source relations before you define views that reference them.
- You must define relations before you define triggers that affect or reference them.

Both source and interactive input are position-independent. You can start or break a statement anywhere, except in the middle of a *lexical unit*. Lexical units are things such as file specifications or words.

For example, **gdef** has no problem processing the following statement:

```
define
field
item_number
char[5];
```

However, this statement does not work, because it breaks a statement in the middle of a lexical unit:

```
define fie
ld
item_number
char[5];
```

Entering DDL Definitions from a File

You can use any text editor to create a file that can be input to **gdef**. For informational purposes, you should use the extension *gdl* to specify a **gdef** file. If you are creating a database, include data definition statements that describe the new database structure.

For example, the following statements in file *emp.gdl* define a database that maintains information about employees:

```
define database "emp.gdb"
  page_size 1024
  {holds employee information};

/* Global Field Definitions */
define field BADGE long;
define field BIRTH_DATE date;
define field DEPARTMENT char [3];
define field FIRST_NAME varying [10];
define field LAST_NAME varying [20];

/* Relation  Definitions */

define relation BADGE_NUM
    BADGE position 0;

define relation DEPARTMENTS
    DEPARTMENT position 0,
```

```
        MANAGER based on BADGE position 1;

    define relation EMPLOYEES
        BADGE position 0,
        BIRTH_DATE position 0,
        FIRST_NAME position 1,
        LAST_NAME position 2,
        SUPERVISOR based on BADGE position 3,
        DEPARTMENT position 4;
```

When you are finished with the data definition statements:

3. Exit from the editor.
4. Invoke **gdef**, giving it the name of the data definition file as an argument:

```
% gdef emp.gdl
```

Gdef reads the source file and builds the system relations. Once the source file is compiled, the database is ready for program access. If **gdef** encounters any errors while compiling the source file, it reports the errors and does not create the database.

Entering DDL Definitions Interactively

If you plan to define only a few entities, you may choose to enter your definitions interactively. When you do this, be sure to end your input with either the end-of-file character for your system, the **quit** command, or the **exit** command.

The following lines show how to enter definitions interactively:

```
% gdef
GDEF> define database "emp.gdb"
CON> page_size 1024
CON> {holds employee information};
GDEF> define field BADGE long;
GDEF> define field BIRTH_DATE date;
GDEF> define field DEPARTMENT char [3];
   ⇓
GDEF> quit
%
```

Gdef reads your input and compiles a list of changes. Once **gdef** has compiled the input, the database is ready for program access.

If **gdef** encounters any syntactic or semantic errors on input, it reports an error. Then, at the end of your input, **gdef** tells you there were errors and asks if you want to continue:

- If you answer y, **gdef** ignores the erroneous actions, creates or modifies the database, and updates the system relations.
- If you answer n, **gdef** does not process your input.

For example:

```
% gdef
GDEF> modify database emp.gdb
standard input:1: expected quoted string, encountered "emp"
GDEF> modify database "emp.gdb";
GDEF> modify field dept varying [3];
standard input:2: expected global field name, encountered "dept"
GDEF> modify field department varying [12];
GDEF> quit
2 errors during input.  Do you want to continue? y
%
```

Using **gdef** interactively is a quick way to define and update metadata. This method is especially useful for tuning your database design.

On the other hand, if you make an error when you use this method, you must retype the whole statement. You also don't have a log of the changes you've made to the database. Because of this limitation, using interactive **gdef** is not the best way to define a new database or to make a lot of changes.

Extracting Metadata

Gdef provides an option that causes it to create a file of data definition statements from an existing database. The **extract** option is useful if you want to see the current metadata for a database that has been greatly modified since it was created.

The following command extracts the metadata from the atlas database to a file called *atlas.gdl*:

Operating System	Command
Apollo AEGIS	gdef -e atlas.gdb atlas.gdl
UNIX	gdef -e atlas.gdb atlas.gdl
VMS	gdef /extract atlas.gdb atlas.gdl

If you omit the output filespec, **gdef** writes the file definition to your current window or terminal.

Maintaining the Database

InterBase provides several database maintenance utilities, which are described in detail in *Database Operations.*

Foremost among the utilities is **gbak**, the backup and restore utility. You should back up your databases on a regular basis, using either **gbak** or the host system backup utility. That way, if your disk drive ever crashes, your work is protected up to the last backup. In any case, if you ever lose your database and you have a backup copy, restore the database from the backup version.

Note

If you use multiple-file databases, you must back up, restore, protect, and copy all the files at once to maintain database consistency.

You gain many advantages by using **gbak** to back up your databases**. Gbak** can:

- Produce a backup that can be restored on any of the supported operating systems, if an appropriate network connection exists. For example, you can use **gbak** to back up a database from a Sun workstation and restore that database to an Apollo workstation.
- Change the database structure, including its page size and file distribution.
- Make unused space in the database available. InterBase automatically reclaims unused space when records are deleted, if the pages containing the deleted records are read. When you use **gbak** to back up your database, **gbak** makes space available, because it directs InterBase to read all database records.

You can also use the **gfix** utility to maintain database performance. The **sweep** option on **gfix**:

- Makes unused space in the database available
- Reduces the amount of time a transaction spends starting up

The file-like aspect of the database allows you to use operating system file copy and delete commands. Therefore, you can do things like copy a database to a scratch file when you test code that might destroy production data.

All users of the database need both read and write access to the database files. However, we recommend that you use host operating system file protection to prevent the accidental or malicious deletion of database files.

Handling Gdef Errors

Gdef reports errors in this format:

```
filename:integer: message
```

The *integer* is the line number on which **gdef** found an error. The *message* explains the **gdef** problem.

For example, the following **gdef** session results in an error being reported for line 1:

```
% gdef
GDEF> modify database "carberry.gdb";
I/O error during "ms_$mapl" operation for file "carberry.gdb"
-name not found (OS/naming server)
standard input:1: Couldn't attach database
GDEF>
```

An error might come from any of three sources:

- **Gdef** itself. These are generally errors that **gdef** encounters when parsing a command such as an unrecognized word or invalid syntax.
- A database error. Database errors can indicate any one of a number of problems. The most likely are the nonexistence of the database specified or privileges that deny you write access. Check the filename or pathname and try again. The example shown above is a database error.
- A bugcheck. Bugchecks reflect software problems that you should report. If you encounter a bugcheck, you should save the error message. Then, submit the error message and the script that led to the bugcheck along with a copy of the database to:

  Interbase Software Corporation
  Customer Support Group
  209 Burlington Road
  Bedford, MA 01730 USA

If you encounter an error and can't decide why there was an error, review the entry for that statement before submitting the bug report.

Most of the messages you receive with **gdef** are self-explanatory. For example, you may reference an object that does not exist, perform some data modification that is not legal, and so on.

For a complete list of **gdef** error messages and their explanations, refer to the *DDL Reference.*

For More Information

For more information on your host operating system's file system and treatment of networks, see the documentation for your operating system.

For more information on creating a database and defining its entities, refer to the remainder of this document and to the following entries in the *DDL Reference*:

- **define database**
- **define field**
- **define filter**
- **define function**
- **define relation**
- **define view**
- **define index**
- **define security_class**
- **define trigger**

Chapter 4
Defining Fields

This chapter discusses how to define fields and describes how and when to use the various field attributes.

Overview

A general discussion of the methods you use to define fields and the characteristics of the attributes you can include is presented below.

Methods for Defining Fields

There are two ways to define fields in **gdef**:

- With a **define field** statement. When you define fields this way, you include them in relations simply by referencing the field names in subsequent **define relation** or **modify relation** statements.
- With a **define** or **modify relation** statement.

No matter which method you use, InterBase stores the field definition as a global definition. This definition is available to any number of relations.

For example, the following statements provide global field definitions that you can use in several relations:

```
define field state char[2];

define relation populations
  population long;
```

Global field definitions provide many advantages because they:

- *Point out natural join paths for high-level programs.* For example, the STATE field occurs in six relations in the atlas database. To display 1950 census data for a state along with other information about that state, you can join the STATES and POPULATIONS relations. You would join these relations by using the STATE field as the join field.
- *Reduce or eliminate divergence among related field definitions.* For example, you can eliminate divergence among the six STATE field definitions when you use the global STATE field definition.
- *Reduce the effort and the error factor involved in changing field types.* For example, if you decide to use a three-letter code for states instead of a two-letter code, you can simultaneously change all relations that use the code by changing the global field STATE.

Field Attributes

There are three types of attributes you can use to describe a field:

- Global attributes that can't be overriden locally. These include the three core attributes:
  - Datatype
  - *Missing value*, which specifies a value that gets returned when a field has no assigned value
  - *Valid if*, which specifies validation criteria for a field

  You can define these attributes by using either the **define field** statement or the **define relation** statement. But you can change them only by using the **modify field** statement.

  The datatype, missing value, and valid if attributes are stored in the RDB$FIELDS system relation.
- Global attributes that can be overriden locally. These include the three **qli**-related attributes:

- — *Edit string*, which specifies a qli display format for a field or computed value
- — *Query header*, which specifies a column header for a qli display
- — *Query name*, which specifies an alternate field name for use in qli

You can define these attributes by using either the **define field** statement or the **define** or **modify relation** statement. You can override them for an individual relation by using a **define** or **modify relation** statement.

The edit string, query header, and query name attributes are stored in both the RDB$FIELDS system relation and the RDB$RELATION_FIELDS system relation.

- A local attribute, security class, that you can define for an individual relation using the **define** or **modify relation** statement.

  The security class attribute is stored in the RDB$RELATION_FIELDS system relation.

Regardless of how you define a field, the only required attributes are its datatype and, if appropriate, its length.

Specifying a Datatype

InterBase supports the following datatypes:

- Binary (integer)
- Float
- Character
- Date
- Blob
- Multi-dimensional array

If your language does not support a particular datatype, InterBase automatically converts the data to an equivalent datatype that can be used in your language. For a list of InterBase datatypes and the name of the corresponding datatype in each of the supported languages, refer to the **define field** entry in the *DDL Reference*.

The range, precision, uses, and restrictions for each of these datatypes are described below.

Specifying Binary Datatypes

InterBase supports two binary datatypes, each of which has an optional scale factor:

- Short (word) has a range of *-32768* to *32767*.
- Long (longword) has a range of *-2**31* to *(2**31)-1*.

You can perform the following operations on the binary datatypes:

- Comparisons. The standard relational operators determine which of two integers is greater, lesser, and so on. Other operators perform string comparisons such as containing, starting with, and matching on numeric fields.
- Arithmetic operations. The standard arithmetic operators determine the sum, difference, product, or dividend of two or more integers.
- Conversions. InterBase automatically converts data between binary, float, and string datatypes. For operations that involve comparisons of numeric data with other datatypes, such as string, InterBase first converts the string data to a numeric datatype and then compares them numerically.
- Sorts.

You also have the option of specifying a *scale factor* for use with the binary datatypes in **qli**, COBOL programs, and PL/I programs. The scale factor is the degree of precision with which InterBase stores data. It is based on a power of 10.

For example, suppose you specify a scale factor of two for a field. This indicates that the degree of precision is in the hundreds. If you then ask **qli** to store the value *167* in that field, **qli** will store the value as *1*.

Now, suppose you specify a scale factor of *negative* two for a field. This indicates that the degree of precision is in the hundredths. If you then ask **qli** to store the value *167* in that field, **qli** will store the value as *167.00*.

A simple guide to how InterBase stores data based on scale factors is presented below:

- For positive scale factors, InterBase divides the input value by 10 to the power indicated by the scale factor. Therefore, using the first example presented above, InterBase divides 167 by 100, yielding the value 1.67. Then, InterBase strips off the fractional digits, yielding the value 1.
- For negative scale factors, InterBase also divides the input value by 10 to the power indicated by the scale factor. Using the second example above, InterBase divides 167 by -100, yielding the value 16700. Then InterBase inserts a decimal point in front of the resulting zeroes, yielding 167.00.

For example, consider the following field definitions in a relation:

```
define database "scale_factor.gdb";
define relation scale_factor
  f1 long,
  f2 long scale 1,
  f3 long scale 2,
  f4 long scale -1,
  f5 long scale -2;
```

The same values were entered for each of the five fields in a record. The input value (no scale factor) is in field F1. The values stored in the database are shown below:

```
  F1            F2           F3           F4           F5
(input value)  (scale 1)   (scale 2)  (scale -1)   (scale -2)
 ===========  =========== =========== ===========  ===========


           1           0           0           10           100
          16           1           0          160          1600
         167          16           1         1670         16700
        1679         167          16        16790        167900
       16791        1679         167       167910       1679100
```

Because **qli**, COBOL, and PL/I support the scale factor, they convert the stored values to their original precision when they retrieve data from the SCALE_FACTOR relation. The following **qli** query prints records from this relation:

```
QLI> ready scale_factor.gdb
QLI> print scale_factor

     F1            F2          F3          F4          F5
(input value)  (scale 1)   (scale 2)  (scale -1)  (scale -2)
 ===========  ==========  ==========  ==========  ==========

          1           0           0         1.0        1.00
         16          10           0        16.0       16.00
        167         160         100       167.0      167.00
       1679        1670        1600      1679.0     1679.00
      16791       16790       16700     16791.0    16791.00

QLI>
```

In comparison, there may be some confusion if you retrieve this data by using programs that don't support the scale factor. For example, consider the following C program that reads records from the SCALE_FACTOR relation:

```
database db = filename "scalefactor.gdb";

{
ready;
start_transaction;
for sf in scale_factor
  printf ("%d %d %d %d %d \n", sf.f1, sf.f2, sf.f3, sf.f4,
     sf.f5));
end_for;

commit;
finish;
}
```

This program produces the following output:

```
     F1            F2          F3          F4          F5
(input value)  (scale 1)   (scale 2)  (scale -1)  (scale -2)
 ===========  ==========  ==========  ==========  ==========

          1           0           0          10         100
         16           1           0         160        1600
        167          16           1        1670       16700
```

```
  1679       167         16       16790        167900
 16791      1679        167      167910       1679100
```

As you can see, the non-scaled F1 is identical for C and **qli**. However, the other fields displayed by C disregard scale factors and retrieve the data as stored.

Specifying Floating Datatypes

InterBase supports two floating datatypes:

- Float, a single precision 32-bit datatype with a precision of approximately 7 decimal digits
- Double, a double precision 64-bit datatype with a precision of approximately 15 decimal digits

```
The following statements  define fields for floating data:

define field salary float;
define field flap_tolerance double;
define field rootbeer float;
```

You can perform the following operations on the floating datatypes:

- Comparisons. The standard relational operators determine which of two numbers is greater, lesser, and so on. Other operators perform string comparisons such as containing, starting with, and matching on the integer portion of floating data.
- Arithmetic operations. The standard arithmetic operators determine the sum, difference, product, or dividend of two or more numbers.
- Conversions. InterBase automatically converts data between binary, float, and string datatypes. For operations that involve comparisons of floating data with other datatypes, such as string and binary, InterBase first converts the data to a numeric datatype and then compares them numerically.
- Sorts.

Specifying String Datatypes

InterBase supports two string datatypes:

- Text or character, which has a range of 1 to 32767 characters
- Varying, which has a range of 1 to 32767 characters

InterBase compresses trailing spaces when it stores text fields. Therefore, a text field that has trailing blanks takes up the same amount of space as an equivalent field defined as varying.

Varying datatypes are only supported in C. For other languages, InterBase converts varying fields to text. InterBase does this by adding spaces to the value in the varying field until the field reaches its maximum length.

The following statements define fields with fixed and varying length data. The bracketed number that follows the field name specifies the maximum length of the field:

```
define field province varying [4];
define field province_name varying [25];
define field capital varying [25];
define field state char [2];
define field zip char [5];
define field name varying [20]
define field type char [1];
define field city varying [25];
define field state char [2];
define field zip char [5];
```

You can perform the following operations on the string datatypes:

- Comparisons. The standard relational operators determine which of two strings is greater, lesser, and so on. Other operators perform string comparisons such as containing, starting with, and matching.

  InterBase uses the ASCII collating sequence for the comparison, thus sorting uppercase before lowercase characters. Therefore, "QUEBEC" sorts before "Quebec," "Quebec" before "quebec," and "ZAPHOD" before "aardvark."

- Limited arithmetic operations. If you try an arithmetic operation on a text field, InterBase first converts the text to a number. This conversion is successful if the text field contains only *numeric* literals; that is, numbers that are stored as text. For example, the fields that make up the degrees and minutes of latitude and longitude in the CITIES relation are stored as char.

  When converting numeric literals to numbers, InterBase reads a period (.) as a decimal point. The input value "943.53" is interpreted as the decimal number "nine hundred forty-three and fifty-three hundreths."

  If the text field contains non-numeric characters other than leading or trailing spaces, or more than a single period, the data conversion fails and the statement containing the arithmetic operation is not executed.

- Conversions. InterBase converts string data to and from all other datatypes except blobs.

- Sorts. The sort is case-sensitive.

Specifying a Date Datatype

Most languages do not support the date datatype. Instead they express dates as strings or structures.

InterBase supports a date datatype that is stored as two longwords. The following statements define date fields:

```
define field standard_date date;
define field ship_date date;
define field halley_comet date;
```

There are two ways to use the date datatype in an InterBase program:

- You can use the **gds_$encode_date** and **gds_$decode_date** routines to convert the date datatype to the UNIX time structure, tm. This is an array of 32-bit words that represents the second, minute, hour, day, month, year, day of week, day in year, and state of daylight savings time.
- You can convert the date datatype to a character string by using **gpre's** casting capability.

For a detailed description of date conversion methods, see the chapter on using date fields in the *Programmer's Guide*.

You can perform the following operations on the date datatype:

- Comparisons. The standard relational operators determine which of two dates is greater, lesser, and so on. Other operators perform string comparisons such as containing, starting with, and matching on date fields.
- Arithmetic operations. You can subtract one date from another. The result is a floating point number that represents the difference in days and fractions of days. You can also add a number to a date, which yields the date plus that number of days. To find an earlier date, you can add a negative number to a date; for example, *"today" + (-4)*.
- Conversions. InterBase converts from date to text and from text to date.
- Sorts.

The range of the date datatype is 1 January 100 to 11 December 5941. If you require dates in the seventh millenium, you should use another datatype. The missing value for date fields is 17 November 1858.

Specifying a Blob Datatype

InterBase supports a datatype called a basic large object or *blob*. A blob looks like a stream or sequential file, but behaves much like a field in a relation. Blobs can hold text, graphics, images, digitized voice, or any other large unstructured data.

Blobs have the following features:

- They are under full transaction and concurrency control.
- They are stored in discrete chunks called *segments*. InterBase provides special calls through GDML and **qli** that let you retrieve and edit individual blob segments.
- You can group the same kinds of blob fields together by subtype. Then you can write blob filters that convert the data from one subtype to another.

Instructions for defining blob segment lengths and subtypes, and for using blob filters are presented below.

Note

You can't index the contents of a blob field or access blob data through SQL statements. You can, however, use blob library routines from an SQL program. These routines are described in the chapter on using blobs in the *Programmer's Guide*.

Defining Blob Segment Lengths

When you define a blob, you can specify a segment length. **Gpre** and **qli** use the segment length to set up a buffer for data transfer between the calling program and InterBase. The segment length you specify does *not* limit the size of the blob or the size of an individual segment.

If you don't specify a segment length, **gpre** and **qli** make the following assumptions:

- **Gpre** assumes a segment length of 80 bytes.
- **Qli** assumes a segment length of 80 bytes.

The following statement defines a blob field and sets a length for each segment:

```
define field blurb blob sub_type text
  segment_length 60;
```

Defining Blob Subtypes

When you define a blob, you can specify a subtype that describes the blob data. For example, you can specify one subtype that holds marked-up text and a second subtype

that holds the output of the text processor. Later on, you can write a blob filter that converts the marked-up text to readable output.

You can use blob subtypes and blob filters to do a large variety of processing. Some other ideas are to:

- Define one blob subtype to hold compressed data and another to hold decompressed data. Then write blob filters that expand and compress blob data.
- Define one blob subtype to hold generic code and other blob subtypes to hold system-specific code. Then write blob filters that add the necessary system-specific variations to the generic code.
- Define one blob subtype to hold word processor input and another to hold word processor output. Then write a blob filter that invokes the word processor.

There are two categories of subtypes you can use:

- Predefined subtypes that InterBase uses internally
- Subtypes that you define as needed

Instructions for using each category of subtypes are presented below.

Using Predefined Subtypes

Table 4-1 shows the predefined subtypes that InterBase uses for internal processing. You can use subtypes 0 and 1 if they are appropriate. The others are for InterBase use only.

Table 4-1. Predefined Blob Subtypes

Blob Subtype	Associated Name	What It Represents
0		An unspecified subtype.
1	text	ASCII text.
2	blr	BLR data. This subtype automatically gets converted to text.
3	acl	InterBase access control lists. This subtype automatically gets converted to text.
4		Reserved for future use.
5		An encoded description of the current metadata for a relation.

Table 4-1. Predefined Blob Subtypes continued

Blob Subtype	Associated Name	What It Represents
6		A description of a multidatabase transaction that finished irregularly. This description includes the transaction's id, host site, database site, and remote site.

For example, the following statement uses **gdef** to define a field that has a predefined subtype of text:

```
define field regular_text blob sub_type text
  segment_length 60;
```

Defining Your Own Subtypes

Unless you use the predefined subtypes 0 or 1 shown in Table 4-1, you should always specify a subtype by using a negative integer. The integer can have a value from -32768 to -1.

By using a negative integer for user-defined subtypes, you avoid conflicts between your subtypes and the subtypes defined by InterBase.

The following statement defines a blob field that has a subtype of -2. This field contains text-processor output:

```
define field nroff_text blob
   sub_type -2;
```

Using Blob Filters

You can use blob filters to convert data from one blob subtype to another. You can access blob filters from any host-language program that contains GDML statements.

Qli automatically uses a blob filter to filter a blob defined with no subtype to text, when asked to display the blob. It also automatically filters blobs defined with subtypes to text, if the appropriate filters have been defined.

To use blob filters, follow these steps:

1. Define the filters to the database.
2. Write the filters and compile them into object code.
3. Create a shared filter library.
4. Make the filter library available to InterBase at runtime.

5. Write an application that requests filtering.

These steps are described in the chapter on using blob fields in the *Programmer's Guide.*

Specifying a Multi-Dimensional Array Datatype

InterBase supports an array datatype that lets you store and retrieve data either as a complete table or a single cell in a table.

You use an array when all three of these conditions exist:

- The data elements naturally form a set. (Individual elements are significant only in the context of the other elements and are all of one datatype.)
- You want to represent and control the entire set of data elements as a single database field.
- You want the capability to identify and access each element individually.

Storing data elements in an array is an alternative to lumping them together in a blob, where they can't be distinguished. It's also an alternative to spreading them out over many fields, where they lose their cohesiveness, are more difficult to maintain as a set, and consume more system resources in overhead than an array.

You can specify up to 16 dimensions for an array and can assign the array one of the following datatypes:

- Short
- Long
- Float
- Double
- Char
- Varying
- Date

The array dimensions must be in the range between -32768 and +32768. If you specify only one boundary for the array, the value of the other boundary defaults to one.

The following statement defines an array that stores data in double format:

```
define field array1 double (20, 1:15, -1:40):
```

This array has three dimensions that have the following ranges:

- The first dimension has a range from 1 to 20.
- The second dimension has a range from 1 to 15.

- The third dimension has a range from -1 to 40.

The following statement defines an array that stores data in text format:

```
define field array2 char[9] (50);
```

This array has one dimension that can hold 50 pieces of data. Each piece of data is char [9].

Array Considerations

You should be aware of the following considerations when you define an array:

- Host languages vary in the ranges they accept. Be sure to define ranges that accommodate all languages that reference the array.
- You can access an array by using embedded GDML statements. You *can't* access an array by using embedded SQL statements or **qli**.

For a detailed discussion of using arrays, refer to the chapter on using array fields in the *Programmer's Guide*.

Defining Field Validation Rules

InterBase supports field-level data validation. You can use a *boolean expression* to restrict the legitimate values for a field.

The following statements define fields with validation expressions, the first stating that prices cannot be negative or zero and the second noting that the listed strings are the only valid ones:

```
define field price long
  valid if (price > 0);

define field standard_flag char[1]
  valid if (
      standard_flag = "Y" or
      standard_flag = "N");
```

Because the validation check can involve any Boolean expression, you can check for equality, inequality, substrings, or ranges of values. You can also check the values of other fields.

For example, the following two fields with interlocking validation might be useful if you add a COUNTRIES relation to the atlas database. (Assume that this relation contains both fields):

```
define field type varying [15]
  valid if (type = 'CAPITALIST', 'SOCIALIST', 'UNDEVELOPED' or
      type missing);
define field sub_type varying [15]
  valid if (
      type = 'CAPITALIST' and
          sub_type = 'LAISSEZFAIRE', 'DEVELOPMENTAL' or
      type = 'SOCIALIST' and
          sub_type = 'MARXIST', 'TROTSKYITE',
              'MAOIST', 'ILL-DEFINED' or
          sub_type missing);
```

Because validation criteria are stored in the database, they eliminate checks in the programs that access the database. InterBase always checks validation expressions when it updates the affected fields.

For more information on data integrity, see Chapter 7, *Preserving Data Integrity*.

Representing Missing Values

InterBase lets any field have a missing value. Rather than storing a value for the field, InterBase sets an internal flag indicating the field has no assigned value. You have the option of declaring an explicit value that InterBase returns if the field value is missing.

For example, consider the PRICE and STANDARD_FLAG fields:

```
define field price long
  valid if (
         price > 0 or
         price missing
  missing_value is -101;

define field standard_flag char[1]
  valid if (
      standard_flag = "Y" or
      standard_flag = "N" or
      standard_flag = "?" or
      standard_flag missing)
  missing_value is "?";
```

These definitions include validation checks that limit acceptable values to the specified values or missing. Both definitions also specify an explicit missing value.

Due to the validation check for PRICE, if someone tries to store a price that is less than or equal to zero, InterBase returns a validation error for the field PRICE and does not store the record in which it is contained, unless the value is exactly -101.

You can enter "Y", "N", or "?" as values for the STANDARD_FLAG field. Records stored with the value "?" can be found using a Boolean test such as *with standard_flag missing*. Even though you stored a question mark and can retrieve a question mark, the Boolean expression *with standard_flag = "?"* does *not* retrieve the records. Should you change the missing value to "N", records that you stored with a question mark will return "N" as the value of STANDARD_FLAG.

The following considerations apply to storing records with explicit missing values. The missing value must:

- Be allowed by a **valid_if** clause. If it is not allowed by this clause, **qli** returns a validation error.
- Meet the datatype specification of the field.

For more information on how the various interfaces handle missing values, see the manual for the interface you are using.

Including Comments

The **comments** clause lets you store comments about the field in a database. A comment can include any of the printable ASCII characters. When you define comments, you use the left bracket ({) and right bracket (}) as delimiters.

The following statements define fields with comments:

```
define field standard_date date { all-purpose date field };

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,
          ⇓
```

Defining a Sequential Number Generator

Use the **gdef define generator** statement to assign sequential numeric keys to a database field. The generator_name must be less than or equal to 31 characters and must be unique across the database.

The following example is a **generator** for invoice numbers, where inv_num is the generator name, create_inv is the trigger name, invoices is the relation name, invoice_-number is the field name, and 1 is the increment:

```
define generator inv_num;
define trigger create_inv for invoices
pre store 0:
begin
    new.invoice_number = gen_id(inv_num,1)
end;
end_trigger;
```

When you use **gdef -e** to extract metadata from a database, the **define generator** statements are also extracted.

Providing Additional Qli Support

Qli is an interactive interface to databases managed by InterBase. You can do the following to make a field more usable in **qli**:

- Define an edit string to format values.
- Define an alternate name for a field.
- Provide an alternative column label for a field.

These options are described below.

Defining Edit Strings

You can define an edit string to specify an alphabetic, numeric, or date format for a field or computed value. The following statement defines a relation with several date fields, each of which has an edit string:

```
define relation family_dates
  name varying [10],
  birth date edit_string "d(2)bm(12)by(4)",
  wedding date edit_string "d(2)'/'n(2)'/'y(4)",
  graduation date edit_string "y(4)";
```

In the preceding date edit strings:

- *d(2)* means "print the day of the month using two characters."
- *b* means "leave a blank space."
- *m(12)* means "print the alphabetic English month using twelve characters."
- *n(2)* means "print the numeric month using two characters."
- *y(4)* means "print the year using four characters."

Qli displays the values as follows:

```
    NAME             BIRTH              WEDDING  GRADUATION
=========== ==================== ========== =========
Katrina     29 May 1956          17/01/1981   1971
Boris       09 April 1954        17/01/1981   1968
```

The following statement defines a relation with edit strings for each of the fields:

```
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 "HHHHHHHHH";
```

In the preceding edit strings:

- The social security number is formatted as it usually appears.
- The telephone number is formatted with the area code in parentheses, a blank, and then the number.
- The salary is displayed in hexadecimal representation.

Qli displays the values as follows:

```
  SOCIAL                    PHONE
 SECURITY    SALARY       NUMBER
=========== ======= ==============
012-34-5678   d9dbb (617) 555-0210
```

Defining Alternate Field Names

If you use a database that was defined by someone else, you may find the definer's choice of field names does not coincide with what your choice might have been. Or if you use **qli** frequently, you may find your field names are not easy to type. For example, although the field names LONGITUDE_DEGREES, LONGITUDE_MINUTES, and LONGITUDE_COMPASS leave no doubt as to the contents of those fields, they are difficult to type.

You can define alternate field names for use in **qli**, thereby allowing you to type shorter, easier, or more meaningful field names. These alternate names are called *query names*.

The following field definitions supply a query name for each of the constituent longitudinal fields:

```
define field longitude_degrees char[2]
  query_name longd;
define field longitude_minutes char[3]
  query_name longm;
define field longitude_compass char[1]
  query_name longc;
```

Therefore, the following **qli** statements are equivalent:

```
QLI> print longitude_degrees of cities

QLI> print longd of cities
```

However, when **qli** displays the records you selected, it uses the full field name to label the column headers. To change the column headers, see below.

Providing Alternative Column Names

For reasons of meaningfulness or length, you may want to have **qli** display a column header other than the field name. For example, if you were to use the full names of the fields that comprise the latitude and longitude of the CITIES as column headers, the display would be very wide and would wrap on smaller screens,

You can either define query headers to match the query name or have entirely different query headers. The following statement defines the same name for both query headers and names:

```
define field longitude_degrees char[2]
  query_name longd
  query_header "longd";
define field longitude_minutes char[3]
  query_name longm
  query_header "longm";
define field longitude_compass char[1]
  query_name longc
  query_header "longc";
```

Note

Query headers must be quoted, while query names are not quoted. The reason for this is that the query header can be a string that contains blanks and multiple words, but the query name is a single word without blanks.

The following statement defines a query header with a blank space:

```
define field longitude_degrees char[2]
  query_name longd
  query_header "long deg";
```

If the query header is much longer than the length of the values displayed (for example, LONGITUDE_COMPASS is one-character long) or too long to fit on one line, you can insert a line break wherever you want. **Qli** uses the slash character (/) as its line break character. For example, the following statement defines a query header that is printed on three lines:

```
define field longitude_degrees char[2]
  query_name longd
  query_header "long-"/"itude"/"degrees";
```

Modifying Fields

There are two ways to modify a field definition:

- Use the **modify field** statement to change field attributes for all relations that use the field.
- Use the **modify relation** statement to change field attributes only in the target relation.

Each of these methods are described below, followed by a list of special considerations.

Using the Modify Field Statement

You can modify the global attributes of a field definition by using the **modify field** statement. Changing the field definition once with this statement changes the field everywhere it occurs.

For example, to change the CITY field from 25 characters to 30, and change the query_header to "Residence," type:

```
modify field city varying[30]
  query_header 'Residence';
```

This statement automatically updates the definitions of the field CITY in RDB$FIELDS, the system relation that defines fields. It also changes the definition of the CITY field in all the relations in which it occurs.

Note

Be careful when you modify a field definition to make the field length shorter. InterBase allows you to do this; but, you may get a truncation error when you try to access the field.

Using the Modify Relation Statement

You can modify the field definition by using the **modify relation** statement. With this statement, you can change the query_name, query_header, edit_string, and security_-class attributes of a field.

When you use the **modify relation** statement, InterBase changes the field attributes only in the target relation. It does not change the field attributes globally.

For example, to modify the query_header and query_name of the CITY field in the STATES relation, type:

```
modify relation states
  modify field city query_header 'Capital'
     query_name 'capital';
```

This statement automatically updates the definition of the field within RDB$RELATION_FIELDS, which describes the characteristics of fields as they occur in an individual relation.

Note

If you try to modify a global attribute of a field by using the **modify relation** statement, **gdef** returns an error.

Considerations for Modifying Fields

When you modify fields, note that:

- Because of its ability to keep track of older record versions, InterBase does not have to make massive updates to the disk when you change a field definition. No matter how many times the definition changes, InterBase can access the appropriate version of the field definition.

  The changes you make will not affect existing users of the database until they reattach to it.

  InterBase always writes to disk the newest version of the field definition when a record is updated.
- You can't change a field's datatype to or from the blob datatype. If you want to do so, define the new field, write a program that copies data to the new field, and then delete the old field.

  For information about reading and writing blobs, refer to the chapter on using blob fields in the *Programmer's Guide*.
- If you change the datatypes or increase the lengths of fields used in a **computed by** field definition, you may have to redefine the **computed by** item to ensure that it's compatible with its source fields. For more information on computed fields, refer to Chapter 5, *Defining Relations*.

Deleting a Field Definition

There are two ways to delete a field definition:

- Use the **delete field** statement to delete a field definition from the database. You can only use this statement when the field is not included in any relations.
- Use the **modify relation** and the **drop field** statements to delete a field definition from a specific relation.

For example, if the CLIMATE field is not included in other relations, you can delete it from the database by typing this statement:

```
modify database "atlas.gdb";
delete field climate;
```

However, if CLIMATE is included in a relation, you must first delete it from the relation in which it's included, and then remove it from the database with a **delete field** statement;

```
modify database "atlas.gdb";
  modify relation cities
      drop field climate;
  delete field climate;
```

If you don't know if a field is included in a relation, use the **qli show field** statement.

For More Information

For more information on defining, modifying, and deleting fields, refer to Chapter 5, *Defining Relations*, and to the following entries in the *DDL Reference*:

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

Chapter 5
Defining Relations

This chapter discusses how to define fields for relations and how to modify and delete relations.

Overview

Relations consist of one or more fields. You define a relation by naming the relation and naming or defining its component fields.

You can create fields in a relation by using:

- **Define relation** statements. These statements can fully define a new field, include a previously defined field, and compute a value from other fields in the relation.
- **Define field** statements, as described in Chapter 4.

Instructions for defining fields with the **define relation** statement are presented below, followed by a discussion of defining external relations and modifying relations.

Defining a New Field for a Relation

You can use the **define relation** statement to define a new field for a relation. For example, the statement below defines the new fields NAME, RANGE, and INHABITED:

```
define relation mountains
  name varying [10],
  range varying [15],
  inhabited char [1]
      valid if (
          inhabited = "Y" or
          inhabited = "N" or
          inhabited missing)
      missing_value is "?";
```

When you define a new field with the **define relation** statement, InterBase stores the field definition as a global template, just as it does when you define a new field with the **define field** statement.

The following considerations apply when you define a field with the **define relation** statement:

- The field name you choose must not conflict with any existing global field names.
- You can reference the field in subsequent relation definitions.
- You can assign any of the attributes discussed in Chapter 4. You can also assign the security class attribute, which is a local attribute.

InterBase stores the definitions of fields defined with the **define relation** statement in the RDB$FIELDS and the RDB$RELATION_FIELDS system relations.

Including Existing Fields in a Relation

You can use the **define relation** statement to include previously defined fields in a relation. For example, the following statements define five new fields with **define field** statements. The **define relation** statement then includes those fields in the BADGE_NUM, DEPARTMENTS, and EMPLOYEES relations:

```
define field BADGE long;
define field BIRTH_DATE date;
define field DEPARTMENT char [3];
define field FIRST_NAME varying [10];
define field LAST_NAME varying [20];

/* Relation  Definitions */

define relation BADGE_NUM
    BADGE position 0;

define relation DEPARTMENTS
    DEPARTMENT position 0,
    MANAGER based on BADGE position 1;

define relation EMPLOYEES
    BADGE position 0,
    BIRTH_DATE position 0,
    FIRST_NAME position 1,
    LAST_NAME position 2,
    SUPERVISOR based on BADGE position 3,
    DEPARTMENT position 4;
```

Whenever you include fields in a relation, you can change certain attributes and assign specific field names. These topics are discussed below.

Changing Field Attributes

You can change the following attributes of an included field:

- Edit string
- Query header
- Query name

You can't change a field's datatype, missing value, or validation criteria.

If any of the relation-specific characteristics conflict with those originally defined for the field, the relation-specific characteristics override those of the field.

For example, a field definition may include a **qli** query name that is inappropriate for a given relation. When you include the field in that relation, you can provide a different query name while retaining other field characteristics.

InterBase stores relation-specific information for an included field in the RDB$RELATION_FIELDS system relation.

Assigning a Specific Field Name

You can give an included field a name specific to a relation. The **based on** clause provides this "alias" capability, as shown in the following example:

```
define relation DEPARTMENTS
    DEPARTMENT position 0,
    MANAGER based on BADGE position 1;
```

The alias you choose must be unique among the field names for the relation, but can duplicate names used in other relations.

InterBase stores both the relation-specific name and the base-field name in the RDB$RELATION_FIELDS system relation.

Defining Computed Fields for a Relation

You can define a field by using a computed formula. A *computed* field is a virtual field. InterBase never stores data in such fields. Instead, it uses the formula to retrieve the requested data.

The following relation contains a computed field in which the author's name is obtained by concatenating the value of FIRST_NAME to LAST_NAME:

```
define relation books
  lib_congress,
  title,
  last_name,
  first_name,
  author computed by (first_name | " " | last_name),
  publisher,
  year,
  city;
```

A computed field can have any combination of the supported arithmetic operations, including addition, subtraction, multiplication, division, and concatenation. For example, this definition of STATES defines two computed fields, ACRES and METRIC_AREA:

```
define relation states
  state,
  capitol,
  area,
  acres computed by (area * 640)
      { 640 acres to a square mile },
  metric_area computed by (area * 2.59)
      { 2.59 square kilometers to a square mile };
```

The following considerations apply to computed fields:

- They are not globally available, because their formula is meaningful only within the context of the relation where they are defined.
- They can have names that duplicate global field names.
- Their names must be unique within the relation in which they are defined.
- If you don't supply a datatype for computed fields, InterBase automatically selects an appropriate datatype, as in the examples above.

InterBase stores the definitions of computed fields in the RDB$FIELDS system relation in the form RDB$*integer* for each computed field. For example, InterBase stores the source name RDB$1 for the first computed field you define.

Defining External Relations

An *external relation* is a relation whose data resides on an external file, rather than in an InterBase database. External relations give InterBase programs and utilities access to files managed by the operating system. They are useful when you want to:

- Make data used by existing non-database applications available to database programs
- Load data created by old applications into the database

You can define external relations that use data residing in Apollo AEGIS stream files, UNIX stream files, VMS RMS sequential files, and VMS RMS indexed files. If you define an external relation for a VMS RMS indexed file, InterBase uses the index. The external file specified in an external relation must reside on the same node as the primary database file with one exception: on VMS, a pathname containing a DECnet specification is permitted for the external file.

Note

You can define external relations only for data composed of fixed-length fields. The fields must be defined with datatypes that InterBase can manage.

Using External Relations

Instructions for using external relations are presented below, followed by a list of special considerations.

Using External Relations for Data Access

To use an external relation for data access, define the relation to InterBase as follows:

1. Name the external file where the data resides.
2. List all the fields contained in the external file. List them in the order in which they appear in the file's records.
3. If the external file contains field separators, include dummy fields in your record definition that map to the separator characters.
4. If the external file contains a record terminator character, define a dummy field with length 1 at the end of the relation.

The following statements define an external relation called EXT_EMPLOYEES that resides on an RMS indexed file:

```
define relation ext_employees external_file
  "$disk2:[home.data.bases]emp.idx"
  badge          long,
  first_name     char [10],
  last_name      char [20],
  supervisor     long;
```

If this file contains record separators and a record terminator character, include dummy fields as appropriate:

```
define relation ext_employees external_file
  "$disk2:[home.data.bases]emp.idx"
  badge          long,
  s1             char [1],
  first_name     char [10],
  s2             char [1],
  last_name      char [20],
  s3             char [1],
  supervisor     long,
  s4             char [1],
  birth_date  char [12],
  t1             char [1];
```

When you reference this relation from **qli** or an InterBase program, InterBase opens the file and reads the records that you request.

Using External Relations for Data Transfer

You can transfer data from an external file to an InterBase relation by using an external relation. Instructions for transferring data, changing the datatypes of loaded data, and converting data formats that InterBase can't interpret are presented below.

Transferring the Data

To use an external relation for data transfer:

1. Define the external relation to InterBase as described in the previous example.
2. Define an internal relation based on the external relation. You can do this easily by using **qli**:

```
QLI> define relation employees based on ext_employees
QLI> show employees
```

```
EMPLOYEES
    BADGE                    long binary
    S1                       text, length 1
    FIRST_NAME               text, length 10
    S2                          text, length 1
    LAST_NAME                text, length 20
    S3                       text, length 1
    SUPERVISOR               long binary
    S4                       text, length 1
    BIRTH_DATE               text, length 12
    T1                       text, length 1
```

3. Drop any unwanted fields from the internal relation:

```
QLI> modify relation employees
CON> drop s1, drop s2, drop s3, drop s4, drop t1
```

4. Use the **qli restructure** statement to copy the data from the external to the internal representation:

```
QLI> employees = ext_employes
```

Changing the Datatypes of Loaded Data

Once you have loaded your data, you may want to change the datatypes of some fields. For example, you may want to change fields that store numbers as ASCII digits to float or double fields. You may also want to change fields that store dates as characters to date fields.

Before you try to change the datatype of a field, be sure the data is in a form InterBase can understand. For example, you can covert date data if it's stored as "MMM DD, YYYY". If the data is not in a form InterBase can understand, you may be able to convert it to an understandable form. An example for doing this is presented below under *Converting Data*. For a list of date formats acceptable to InterBase, refer to the chapter on using date fields in the *Programmer's Guide*.

When you change the datatypes of loaded data, you must first separate the field definitions in the exernal file from the internal definitions. This is because you can't change a datatype in an external file. For example, to change the datatype of the BIRTH_DATE field:

1. Define a new global field that describes the external data:

```
QLI> define field old_bd char [12]
```

2. Redefine the external field so it uses the new global field:

```
QLI> modify relation ext_employees
CON> modify field birth_date based on old_bd
```

3. Change the global field definition for BIRTH_DATE:

```
QLI> modify field birth_date date
```

InterBase converts your data automatically.

Converting Data

Sometimes data in external files is not in a format that InterBase can interpret and convert automatically. This is often true for date data. For example, suppose the BIRTH_DATE field in the example above contains the value "760704" and that this value represents July 4, 1976.

To convert this data:

1. Define three two-character fields to represent the date in the external relation:

```
define relation ext_employees external_file
  "$disk2:[home.data.bases]emp.idx"
  badge          long,
  s1             char [1],
  first_name     char [10],
  s2             char [1],
  last_name      char [20],
  s2             char [1],
  supervisor     long,
  year           char [2],
  month          char [2],
  day            char[2],
  t1             char [1];
```

2. Define an internal relation based on the external relation:

```
QLI> define relation employees based on ext_employees
```

3. Drop any unwanted fields from the internal relation:

```
QLI> modify relation employees
CON> drop s1, drop s2, drop s3, drop s4, drop t1
```

4. Use the **qli restructure** statement to copy the data from the external to the internal representation:

```
QLI> employees = ext_employes
```

5. Add a BIRTH_DATE field to the internal relation:

```
QLI> modify relation employees
CON>  add field birth_date date
```

6. Convert the data in the internal relation:

```
QLI> for employees modify using
CON> birth_date = day | "/" | month | "/" |year
```

7. Drop the YEAR, MONTH, and DAY fields from the internal relation:

```
QLI> modify relation employees
CON> drop field year, drop field day, drop field month
```

If you plan to move data regularly from external to internal format, you should use a trigger to build the new date format from the old year, month, and day. In this case, the internal relation should include the YEAR, MONTH, DAY, and BIRTH_DATE fields, so that data can be tranferred from the old format to the new one.

Considerations for Defining and Using External Relations

The following considerations apply to the use of external relations:

- The data in the external file must be defined with datatypes that InterBase can manage. For example, the data can be ASCII character, integer, or float, but not packed decimal or EBCDIC.
- The fields in the external file must be of fixed length. If the data contains seperator characters, you must include dummy fields in your definition of the external relation. These fields must map to the separator characters.
- You can use InterBase to add records to an external relation, but you can't modify or delete the data in existing records.
- External relations are not under transaction control. Commit and rollback operations do not affect updates to external relations, so that uncommitted updates are visible immediately to other users.
- InterBase's multi-generational concurrency control does not apply to external relations. Instead, the external relation uses operating-system level concurrency control. For some operating environments, only one user is allowed to read or write to an external file at any time. For other environments, users can share the file, but the file is not protected from concurrent updates.
- The metadata for external files can't be changed dynamically. To add or drop fields from an external relation, you must create a new file with the structure you want and restructure the data from the old file to the new file.

Because the concurrency and consistency control provided by InterBase is significantly better than that provided by the operating systems, you should load external data into the database if you plan to update that data.

If you plan to read the data only, you can leave it as an external relation.

Modifying Relations

You can change a relation by:

- Adding fields
- Dropping fields
- Changing the query name, query header, edit string, or security class of a field within the relation
- Changing the default display order of fields within the relation
- Changing the security class associated with the relation

Changes made with a **modify field** statement to global field characteristics (datatype, missing value, and validation criteria) are automatically reflected in each relation in which those fields occur.

The following **modify relation** statement modifies the CITIES relation by a dropping a field, adding fields, and adding query names:

```
modify relation cities
  drop field population,
  add field population_1950 long,
  add field population_1960 long,
  add field population_1970 long,
  add field population_1980 long,
  modify field latitude_degrees
      query_name ld,
  modify field latitude_minutes
      query_name lm,
  modify field latitude_compass
      query_name lc;
```

This **modify relation** statement changes the default **qli** display order of fields in the CITIES relation. It moves the POPULATION fields to the beginning of the display:

```
modify relation cities
  modify field population_1950 position 1,
  modify field population_1960 position 2,
  modify field population_1970 position 3,
  modify field population_1980 position 4,
  modify field city position 5;
  modify state position 6,
  modify field altitude 7,
```

Special considerations are involved in modifying global field characteristics and computed fields. These are described below.

Modifying Global Field Characteristics

You can't change global field characteristics through a relation, even though you may have originally defined the field through the relation.

When you want to change any field characteristic other than its query name, query header, position, description, or security class, you must use a **modify field** statement.

Modifying a Computed Field

You can't directly modify the definition of a computed field. Instead, you must redefine the field by dropping it and and re-adding it.

If you have a computed field that consists of concatenated fields, you need to redefine the computed field whenever you increase the length of the constituent fields.

For example, suppose you had a field, FULL_NAME, computed by concatenating a 10-character field FIRST_NAME to a space and a 15-character field LAST_NAME:

```
define relation employee
  first_name varying[10],
  last_name varying[15],
  full_name computed by
      (first_name | " " | last_name),
  emp_id char[9],
  dept_num char[3];
```

Gdef provides a length for the field definition by adding the length of the fields with the length of the separating space. In this example, the length of FULL_NAME is 26.

Suppose you modify the source fields later on by increasing their length to 15 and 25. Although the concatenation now results in a field that's 41 characters long, the length of the computed field remains 26. This problem does not surface until you reference the computed field, at which time InterBase returns a data conversion error.

You don't have to redefine a computed field if you decrease the length of the constituent fields, or if the fields are computed numerically.

Deleting Relations

You delete a relation by using the **delete relation** statement. This statement removes the specified relation and *all of its data* from the database. You can't roll back this operation, so use it carefully.

The following statement deletes the MOUNTAIN_RANGES relation from the database:

```
delete relation mountain_ranges;
```

Note

You can't delete a relation that's used in a view. You must delete the view before you delete the relation.

For More Information

For more information on defining, modifying, and deleting relations, refer to these entries in the *DDL Reference*:

- **define relation**
- **modify relation**
- **delete**
- Boolean expression

For more information on defining fields for relations, refer to Chapter 4, *Defining Fields*, and to these entries in the *DDL Reference*:

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

Chapter 6
Defining Views and Indexes

This chapter discusses how to define, modify, and delete views and indexes. It also discusses how to add indexes to existing relations.

Overview

A *view* is a relation that consists only of a stored definition rather than stored data. A view derives data from one or more source relations each time it's requested. To a user, a view is indistinguishable from a regular stored relation.

You use the following statements to define, modify, and delete a view:

- define view
- modify view
- delete view

An *index* is an internal data structure that InterBase uses to locate records quickly. When executing a query, InterBase first looks to see which, if any, indexes exist for the

involved relations. It then calculates the best way to answer the query and uses the indexes that yield the best performance.

You can define any number of indexes on a relation, each of which will perform with the same quick speed.

You use the following statements to define, modify, and delete an index:

- define index
- modify index
- delete index

Detailed instructions for defining, modifying, and deleting views and indexes are presented in the remainder of this chapter.

Defining Views

A view can be:

- A vertical subset of fields from a single relation. This type of view limits the fields that are displayed.
- A horizontal subset of records from a single relation. This type of view limits the records that are displayed.
- A combined vertical and horizontal subset of records from a single relation. This type of view limits both the fields and records that are displayed.
- A subset of records from many relations. This type of view usually performs a join operation.

Examples of views are presented below, followed by special considerations.

Limiting Fields

You can define a view containing only those fields your program references. The following view contains several fields from all records in the CITIES relation:

```
define view map_cities of c in cities
  { data for locating cities on map grid }
  c.city,
  c.state,
  c.latitude,
  c.longitude;
```

The phrase *c in cities* is a *record selection expression* (RSE) that specifies which records are to be returned. In this case, all records in the CITIES relation qualify.

Limiting Records

You can define a view that limits the number of returned records. The following view returns all fields for cities located above 40 degrees latitude:

```
define view ice_belt of c in cities
  with c.latitude_degrees ge 40
      c.city
      c.state,
      c.population,
      c.altitude,
      c.latitude_degrees,
      c.latitude_minutes,
```

```
        c.latitude_compass,
        c.longitude_degrees,
        c.latitude_minutes,
        c.longitude_compass,
        c.latitude,
        c.longitude;
```

Limiting Both Fields and Records

You can define a view that limits both the fields and the records returned to the program. The following view returns a subset of fields for cities located above 40 degrees latitude:

```
define view ice_belt of c in cities
  with c.latitude_degrees ge 40
      c.city,
      c.state,
      c.population,
      c.altitude,
      c.latitude,
      c.longitude;
```

Accessing Records from Multiple Relations

You can use a view to access data from multiple relations. This access can take many forms. For example, you can use a view to:

- Retrieve data from multiple relations.
- Check other relations for the existence of specific records. In this case, the view may or may not display data from the other relations.
- Calculate the value of a computed field.

Examples of these uses are shown below.

Example 1 — Retrieving Data from Multiple Relations

This view joins the CITIES relation to the STATES relation:

```
define view capital_cities of s in states
  cross c in cities with
      s.capital = c.city and
      s.state = c.state
          c.city,
```

```
    s.state_name,
    c.altitude,
    c.latitude,
    c.longitude;
```

You can use as many **cross** clauses as necessary to select the records you need.

Example 2 — Checking for the Existence of Specific Records

This view returns only those STATES records for which there is at least one record in the SKI_AREAS relation that has the same value for the STATE field:

```
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;
```

Example 3 — Calculating the Value of a Computed Field

You can have InterBase compute the value of a field by referencing field values in other relations.

This view includes several fields that are computed by dividing field values in one relation by field values in another relation:

```
define view population_density of s in states
  cross p in populations with s.state = p.state
      s.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);
```

In this example, InterBase calculates the values for the various density fields by taking the census data from the POPULATIONS relation and dividing it by the area of each state.

Considerations for Defining Views

- Views that don't have join operations can be updated like regular stored relations. However, multi-relation views and single-relation views based on reflexive joins can be updated only through triggers.

  For more information on updating views, refer to Chapter 7, *Preserving Data Integrity*.

- You can't include a record selection expression *sorted clause* in a view definition.
- SQL security does not apply to views. If you want to secure a view, assign it an appropriate security class. For more information on securing views, refer to Chapter 8, *Securing Data and Metadata*.

Modifying and Deleting Views

You modify a view by using the **modify view** statement. This statement lets you drop a field from a view, and add or change a security class.

The following statement makes several changes to the GEO_CITIES view:

```
modify view geo_cities
   {new version of the geo_cities view }
   drop field altitude,
   drop security_class,
   add security_class top_secret;
```

If you want to change record selection criteria or add fields to a view, you must delete the view and then re-create it with the new criteria and fields.

You delete a view by using the **delete view** statement. The following statements delete the GEO_CITIES view and recreate it with a new field:

```
delete view geo_cities;

define view geo_cities of cities
   { comment goes here }
   altitude,
   latitude,
   longitude,
   security_class top_secret;
```

Defining Indexes

As a general rule, you should define an index for:

- A relation's primary key. You should always use the **unique** option when you define an index for a primary key. If the key consists of multiple fields, you would define a *multi-segment* index to represent it.
- A relation's foreign keys. Placing an index on primary and foreign keys enhances performance when relations are joined. It also preserves referential integrity.
- Non-key fields that are accessed frequently for retrieval purposes.
- Non-key fields that are unique.

You should *not* define an index for non-key fields that are updated frequently. Updating an indexed field takes longer than a non-indexed field, because InterBase must modify the index whenever the field value changes.

Because indexes require additional storage space, you should only define them where needed.

Index Definition Examples

You define an index by using the **define index** statement. For example, the following statement defines a single-segment index for the STATE field in the STATES relation:

```
define index states_idx1 for states
  state;
```

This statement defines a multi-segment index for the STATE and RIVER fields in the RIVER_STATES relation:

```
define index rivstate_indx1 for river_states state, river;
```

You can use indexes to eliminate duplicates and to make a descending sort more efficient. If you want to:

- Eliminate duplicates, include the **unique** option when you create the index.

  This causes InterBase to disallow users from storing duplicate values in the indexed field. For example:

```
define index states_idx1 for states
   unique state;

define index states_idx2 for cities
   unique city, state;
```

- Make a descending sort on a field or group of fields more efficient, include the **descending** option when you create the index. For example:

```
define index states_idx1 for states
   unique descending state;
```

- Make an ascending sort on a field or group of fields more efficient, either include the **ascending** option when you create the index or leave off the **ascending** or **descending** options entirely. The **ascending** option is the default. For example:

```
define index states_idx1 for states
   unique ascending state;

define index states_idx2 for states
   unique state;
```

Considerations for Defining Multi-Segment Indexes

InterBase optimizes a query against a multi-segment index by using the first field in the index. If the query uses an index segment without using all previous segments, InterBase does not optimize the query.

For example, suppose you define the following index:

```
define index states_idx2 for cities
   unique city, state;
```

This query is optimized against the index:

```
select * from cities where city = 'Boston';
```

This query is not optimized against the index:

```
select * from cities where state = 'MA';
```

Because of the way InterBase optimizes queries against multi-segment indexes, it's best to use several single-segment indexes when you expect to issue a mixture of queries against a relation. The main reason to use a multi-segment index is to enforce uniqueness.

Modifying Indexes

You can do any of the following when you modify indexes:

- Change an index's **unique** option.
- Change an index's **ascending** or **descending** option.
- Deactivate an active index. This action is recommended before loading a large number of records, so that the index is not modified incrementally.
- Reactivate an inactive index.

For example, this statement changes an index to inactive:

```
modify index idx_100 inactive;
```

Special Considerations for Indexes

The following considerations apply to indexes:

- If you want to change an index to include another key or to remove an existing one, you must delete the index and create a new one.
- You can't add a **unique** option to an index if the indexed field contains duplicate data.
- If you try to modify an index that's currently in use in an active database, **gdef** waits until the index is not in use.
- Indexes can get unevenly filled when you make a lot of changes to your database. There are two ways to reclaim unused index space and reduce the depth of the index tree:
  - — Deactivate and reactivate the affected indexes.
  - — Back up and restore your database using **gbak**.

  Both of these actions cause the indexes to be rebuilt. For information on using **gbak**, refer to the chapter on backup and recovery in *Database Operations*.

Deleting and Adding Indexes

You delete an index by using the **delete index** statement. For example, the following sequence of statements first deletes an index named IDX_1 and then creates a new IDX_1:

```
delete index idx_1;

define index idx_1 for cities
  unique state, city;
```

You can't delete an index that's currently in use in an active database. If you try to do this, the results depend on the type of transaction you are using. If the transaction is a:

- Wait transaction, the delete waits until the index is not in use. Both **gdef** and **qli** use wait transactions.
- Nowait transaction, **gdef** returns an error.

As a rule, you create new indexes for a database more than you delete existing ones. For example, during your initial application analysis you might not have realized that many queries will be referencing an unindexed field. You can probably increase performance significantly by defining an index for that field.

Creating an index for an existing relation may take some time, because **gdef** builds the index before it exits.

For More Information

For more information on defining, modifying, and deleting views, refer to these entries in the *DDL Reference*:

- **define view**
- **modify view**
- **delete**
- Boolean expression

For more information on defining, modifying, and deleting indexes and more information on adding new indexes, refer to these entries in the *DDL Reference*:

- **define index**
- **modify index**
- **delete**

Chapter 7
Preserving Data Integrity

InterBase provides a variety of mechanisms that automatically preserve the integrity of your data. This chapter discusses integrity rules and the InterBase mechanisms for enforcing those rules.

Overview

There are four types of rules that can be used to preserve data integrity:

- Entity integrity
- Referential integrity
- Domain integrity
- Application integrity

Entity Integrity

Entity integrity preserves the uniqueness of each relation in your database. This rule states that the following conditions are true for every database relation:

- Each primary key is unique.
- Each component of a primary key in a base relation is non-null.

You enforce entity integrity by defining a unique index on the fields that comprise the primary key. Instructions for defining a unique index are presented later in this chapter.

Referential Integrity

Referential integrity protects the relationship between associated database relations when the relations are updated. It ensures that one of the following conditions is true for a relation that contains a foreign key.

- The value of the foreign key references an existing primary key.
- The value of the foreign key is null.

You derive referential integrity rules from your company's business requirements. For example, you can define a referential integrity rule to prevent an atlas database application from adding a city for which no valid state exists.

You enforce referential integrity by using triggers to specify the relationship between two relations. For example, you can write a trigger that automatically checks whether a state exists in the STATES relation before a new city is stored in the CITIES relation.

Instructions for using triggers are presented later in this chapter.

Domain Integrity

Domain integrity ensures that the data inserted into relations is valid. You can enforce domain integrity at two different levels:

- By defining a field and assigning it a datatype, you authorize the database to restrict the insertion of data to that datatype. For example, a field defined as a double float does not accept a character string.
- You can then define validation criteria that further restricts the values that can be inserted into the field. For example, you can restrict a field so that it accepts the abbreviation MA, but rejects Ma.

Instructions for defining validation criteria are presented later in this chapter. For information on assigning datatypes to a field, refer to Chapter 4, *Defining Fields*.

Application Integrity

Application integrity ensures that the data in your database meets all your company's business requirements. This is the most general form of data integrity. It includes the specific types of data integrity discussed above (entity integrity, referential integrity, and domain integrity). It also includes all other business requirements.

Not allowing users to delete or modify cities that are state capitals is an example of application integrity.

You can enforce most application integrity rules by using triggers.

Defining a Unique Index

You define a unique index by using the **unique** option on the **define index** statement. For example, to define a primary key for the CITIES relation, you would define a unique index on the CITY and STATE fields:

```
define index cities_1 for cities unique
  city,
  state;
```

This definition disallows all CITIES records that duplicate a city by name and state.

Once you define a unique index, InterBase automatically enforces the uniqueness of that index. User programs don't need to check for duplicates. Instead, they need to include error-handling code that can handle a user's attempt to store a duplicate record.

Note

You can't define a unique index for fields containing null values. You also can't store null values into fields that are part of a unique index.

For more information on defining indexes, refer to Chapter 6, *Defining Views and Indexes.*

Defining Validation Criteria

You define validation criteria for a field by using the **valid_if** clause on the **define field** statement. For example, to restrict the ELECT_APPT field's value to "y", "n", or missing, type:

```
define field elect_appt char [1]
    valid if (elect_appt = 'y' or elect_appt = 'n' or
      elect_appt missing);
```

To restrict the CITY field to non-null values, type:

```
define field city varying [25]
    valid if (city not missing);
```

InterBase automatically enforces validation criteria when it updates a field. For more information on defining validation criteria, refer to Chapter 4, *Defining Fields.*

Using Triggers

A *trigger* is a piece of code that executes a specific action when a record in a relation is stored, modified, or erased. Because triggers can access other relations, they can provide both referential integrity and application integrity.

Triggers are automatically executed, regardless of the interface through which you store, modify, or erase the associated records.

Defining a Trigger

You define a trigger by using:

- Any *embedded GDML statement*, except for:
  - The **on_error** statement
  - All blob handling statements other than assignments
- *GDML trigger language extensions,* including the following predefined context variables:
  - **New,** which refers to the newly created record.
  - **Old**, which refers to the record being modified or erased. For a complete description of these extensions, refer to the **define trigger** statement in the *DDL Reference*
- A *trigger definition statement* that names the trigger and associates it with a relation.
- A *trigger activity indicator* that specifies whether a trigger is active or inactive. An active trigger executes immediately after the current transaction ends. By default, all triggers are active.
- Three other trigger indicators:
  - A *time indicator* that specifies whether the trigger is activated before (**pre**) or after (**post**) the associated store, modify, or erase action.
  - An *action indicator* that associates the trigger with a store, modify, or erase action.
  - A *sequence indicator* that groups triggers together and specifies when they are executed in relation to other groups of triggers. This only applies when you define multiple triggers of the same type for the same relation.

    For example, if you assign two *pre* modify triggers a sequence number of 0 and three other *pre* modify triggers a sequence number of 1, the first group executes before the second. Within each group, the order of execution is random.

    If you don't assign a sequence indicator, the trigger is given an indicator of 0.

- The ***abort*** *statement,* which terminates the action that called the trigger and returns a status code and message to the fourth longword of the status vector.
- A *message definition statement* that lets you associate a message with an abort code. The message is printed in **qli** and is available through the **gds_$print_status** routine in a GDML or SQL program, if the program fails with the associated code.

  You can define multiple messages for each trigger.

Trigger Definition Components

The example below shows the various components of a trigger definition. It defines a trigger that prevents users from modifying a state capital.

Trigger definition statement

```
define trigger retain_capitals for cities
```

Activity indicator

```
active
```

Time, action, and sequence indicators

```
pre modify 0:
```

Trigger logic

```
begin
if (new.city ne old.city)
  or (new.state ne old.state)

  begin
    for st in states with st.capital = old.city
    and st.state = old.state
    abort 1;
    end_for;
  end;

end;

end trigger
```

Message definition statement

```
message 1:"You can't modify a state capital.";
```

The trigger definition components are described below:

- *Trigger definition statement.* The trigger named RETAIN_CAPITALS is associated with the cities relation.
- *Activity indicator.* This trigger is active. Because it's a *pre* type trigger, this trigger will execute before the associated operation takes place.
- *Time, action, and sequence indicators.* This trigger will be activated before the modify operation. It will be the first modify trigger activated at this time.
- *Trigger logic.* If the user has modified the CITY or STATE field, execute the logic that follows.

  If the CITY field being modified is a state capital, then stop the modify and return the message associated with message code 1.
- *Message definition statement.* Add this message to the trigger and associate it with message code 1.

Defining Multiple Triggers

You can define as many triggers as you need for a particular relation. You can define these triggers in any combination.

For example, you may choose to define a single store, modify, and erase trigger for a particular relation. Or, you may choose to define multiple store, modify, and erase triggers for that relation.

The ability to define multiple triggers provides the following advantages:

- It gives you greater control over when the trigger is activated. For example, you can define one or more triggers that get activated before a particular operation, and one or more triggers that get activated after that operation.
- It lets you write several simple triggers rather than one elaborate one. This simplifies the writing, testing, and maintaining of the trigger code.

Example of Defining Multiple Triggers

The example below shows how to define two modify triggers for a relation:

- The first trigger prevents users from modifying state capitals. This trigger is activated before the modify operation.

- The second trigger enforces referential integrity between the CITIES relation and the TOURISM relation. It ensures that the city name and state code in the TOURISM relation are modified when the corresponding city name and state code in the CITIES relation are modified. This trigger is activated after the originial modify operation.

```
define trigger retain_capitals for cities
pre modify 0:
  begin
  if (new.city ne old.city)
    or (new.state ne old.state)
    begin
    for st in states with st.capital = old.city and
      st.state = old.state
    abort 1;
    end_for;
    end;
  end;
end_trigger
message 1: "You can't modify a state capital";

define trigger enforce_integrity for cities
post modify 0:
  begin
    if (new.city ne old.city)
      or (new.state ne old.state)
    begin
    for t in tourism with t.city = old.city and
       t.state = old.state
    modify t using
      t.city = new.city;
      t.state = new.state;
    end_modify;
    end_for;
    end;
  end;
end_trigger;
```

Using Triggers with Views

You can use triggers to enforce referential and application integrity rules for views in the same way that you use them to enforce these rules for stored relations.

You *must* use triggers if you want to update views that are based on multi-relation or reflexive joins. Views that do not include joins can be updated like regular relations.

Example of Using Triggers with Views

Consider a view of the CITIES and STATES relations that displays the name of each large city and its corresponding state:

```
define view big_cities of
  c in cities cross s in states over state
  with c.population > 100000
    c.city,
    c.population,
    s.state_name;
```

The trigger below allows a user to store a new city through the BIG_CITIES view. This trigger enforces the following integrity rules:

- When a user stores a new city through the BIG_CITIES view, a new record is created in the CITIES relation.
- The state must already exist in the STATES relation for the store to take place.

```
define trigger update_through_view for big_cities
pre store 0:
begin
  if not any s in states with s.state_name = new.state_name
    abort 1;
  else if not any c in cities with new.city = c.city and
    c.state = first s.state from s in states
    with s.state_name = new.state_name
    then store c in cities using
      c.city = new.city;
      c.population = new.population;
      c.state = first s.state from s in states with
        s.state_name = new.state_name;
    end_store;
end;
end_trigger
message 1: "State name is invalid.";
```

Updating a Trigger Definition

There are three ways of updating a trigger definition. You can:

- Modify the definition
- Delete the definition
- Deactivate the trigger

Instructions for performing these actions are presented below. The associated examples are based on this trigger definition:

```
define trigger retain_capitals for cities
pre modify 0:
begin
  if (new.city ne old.city)
    and (new.state ne old.state)
  begin
    for st in states with st.capital = old.city
      and st.state = old.state
    abort 1;
    end_for;
  end;
end;
end_trigger
message 1: "You can't modify a state capital.";
```

Modifying a Trigger Definition

You modify a trigger definition by using the **modify trigger** statement. For example, to remove the **if** statement from the RETAIN_CAPITALS trigger, type:

```
modify trigger retain_capitals
begin
  for st in states with st.capital = old.city
    and st.state = old.state
    abort 1;
  end_for;
end;
end_trigger;
```

To change the sequence number of RETAIN_CAPITALS, type:

```
modify trigger retain_capitals
pre modify 2:
end_trigger;
```

To modify the trigger message, type:

```
modify trigger retain_capitals
msgmodify 1: "A state capital can't be modified.";
```

Deleting a Trigger Definition

You delete a trigger definition by using the **delete trigger** statement. For example, to delete the RETAIN_CAPITALS definition, type:

```
delete trigger retain_capitals;
```

Deactivating a Trigger

You deactivate a trigger by setting the trigger activity indicator to inactive. Once a trigger is deactivated, it remains that way until you reset the indicator to active.

For example, to deactivate the RETAIN_CAPITALS trigger, type:

```
modify trigger retain_capitals
inactive;
```

To reactivate this trigger, type:

```
modify trigger retain_capitals
active;
```

Special Considerations for Triggers

When you use triggers, you need to consider whether a trigger is looking up data in another table. If the lookup table is large and/or the trigger will be fired frequently, consider placing an index on the lookup field to make the trigger processing more efficient.

When you use triggers, you also need to consider:

- What happens when a trigger or an operation associated with a trigger aborts
- How triggers work with InterBase's transaction processing
- What happens when the action executed by one trigger fires off another trigger

Each of these considerations is discussed below.

Undoing Triggers

InterBase treats a database operation and its associated triggers as a single unit. If any part of this unit aborts, InterBase automatically rolls back all associated changes:

- If a *pre* type trigger aborts, InterBase rolls back the actions performed by the associated *pre* type triggers. It does not perform the store, modify, or erase operation.
- If the operation aborts, InterBase rolls back the actions performed by the associated *pre* type triggers. It does not execute the *post* type triggers.
- If a *post* type trigger aborts, InterBase rolls back the actions performed by the associated *post* type triggers; the store, modify, or erase operation; and the actions performed by the associated *pre* type triggers.

Transaction Processing

Triggers operate within the context of transaction processing. Regardless of whether a set of triggers complete successfully or not, the programmer can decide whether to commit or roll back a transaction at a later point in time.

For example, suppose a program has the following logic:

1. Start a transaction.
2. Request a modify operation.
   a. The *pre* modify triggers execute.
   b. The modify operation is performed.
   c. The *post* modify triggers execute.
3. Perform other actions.
4. Commit or rollback the transaction.

Here are some possible outcomes to this scenario:

- Steps 2 and 3 complete successfully and the programmer issues a commit in Step 4. In this case, all database changes are saved.
- Steps 2 and 3 complete successfully and the programmer issues a rollback in Step 4. In this case, all database changes are rolled back.
- Step 2 fails, Step 3 completes successfully, and the programmer issues a commit in Step 4. In this case, all database changes made in Step 3 are committed. Any database changes made in Step 2 have already been rolled back.

- Step 2 completes successfully, Step 3 fails, and the programmer issues a rollback in Step 4. In this case, all database changes made in Steps 2 and 3 are rolled back.

Trigger Interrelationships

Whenever you define a trigger, you should be aware that other triggers might be automatically fired off when the first trigger is executed. For example, suppose you define two triggers:

- A modify trigger for the CITIES relation that cascades an update to the SKI_AREAS table.
- A modify trigger for the SKI_AREAS relation that checks to see if the new city is valid before it modifies a city.

If you make the trigger on the CITIES relation a *pre* type trigger, the trigger won't do what you want it to do. This is because:

1. InterBase tries to update the city field in the SKI_AREAS relation *before* that city has been stored in the CITIES relation.
2. When InterBase tries to update the city field in the SKI_AREAS relation, the modify trigger on that relation is fired.
3. This trigger checks to see if the CITIES relation contains the new city. Because it doesn't yet contain this new city, the trigger aborts.
4. As a result of the trigger failure, the original modify operation fails.

If you make the trigger on the CITIES relation a *post* type trigger, both triggers will be successful.

The correct definitions for these two triggers are shown below:

```
define trigger cascading_modify for cities
post modify 0:
begin
  if (new.city ne old.city)
    or (new.state ne old.state)
  begin
    for t in ski_areas with t.city = old.city
      and t.state = old.state
    modify t using
      t.city = new.city,
      t.state = new.state;
    end_modify;
    end_for;
  end;
end;
```

```
end_trigger;

define trigger valid_modify for ski_areas
pre modify 0:
  begin
    if not any c in cities with c.city = new.city
    and c.state = new.state
    abort 1;
   end;
end_trigger
message 1: "City name is invalid.";
```

More Trigger Examples

Examples that show various uses of triggers are presented below.

Example 1 — Storing a Foreign Key

This example shows how to enforce referential integrity when storing a foreign key. In this example, a user can only store a new TOURISM record if the record contains a valid city. This city must already exist in the CITIES relation.

```
define trigger foreign_key_store for tourism
pre store 0:
  begin
    if not any c in cities with c.city = new.city
    and c.state = new.state
    abort 1;
   end;
end_trigger
message 1: "City name is invalid.";
```

Example 2 — Implementing a Cascading Delete

This example shows how to enforce referential integrity by implementing a cascading delete. In this example, when a user deletes a city from the CITIES relation, that city is also deleted from the TOURISM and SKI_AREAS relations.

```
define trigger cascading_delete for cities
post erase 0:
  begin
    for t in tourism with t.city = old.city
    and t.state = old.state
```

```
    erase t;
    end_for;
    for s in ski_areas with s.city = old.city
    and s.state = old.state
    erase s;
    end_for;
   end;
  end_trigger;
```

Example 3 — Implementing a Restricting Delete

This example shows how to enforce referential integrity by implementing a restricting delete. In this example, a user can only delete a city from the CITIES relation if that city is *not* a foreign key in the TOURISM and SKI_AREAS relations.

```
define trigger restricting_delete for cities
post erase 0:
  begin
    for t in tourism with t.city = old.city
    and t.state = old.state
    abort 1;
    end_for;
    for s in ski_areas with s.city = old.city
    and s.state = old.state
    abort 1;
    end_for;
   end;
  end_trigger
message 1: "This city cannot be deleted, because it exists in
other relations";
```

Example 4 — Implementing a Nullifying Delete

This example shows how to enforce referential integrity by implementing a nullifying delete. In this example, when a user deletes a city from the CITIES relation, the same city is set to null in the TOURISM and SKI_AREAS relations.

```
define trigger nullifying_delete for cities
post erase 0:
  begin
    for t in tourism with t.city = old.city
    and t.state = old.state
    modify t using
      t.city = null;
```

```
    end_modify;
    end_for;
    for s in ski_areas with s.city = old.city
    and s.state = old.state
    modify s using
      s.city = null;
    end_modify;
    end_for;
   end;
  end_trigger;
```

Example 5 — Returning Multiple Messages

This example shows how to define a trigger that returns multiple messages. This example expands on Example 3, *Implementing a Restricting Delete.*

```
define trigger restricting_delete for cities
pre erase 0:
  begin
    for t in tourism with t.city = old.city
    and t.state = old.state
    abort 1;
    end_for;
    for s in ski_areas with s.city = old.city
    and s.state = old.state
    abort 2;
    end_for;
   end;
  end_trigger
message 1: "This city cannot be deleted, because it exists
  in the TOURISM relation",
message 2: "This city cannot be deleted, because it exists
  in the SKI_AREAS relation.";
```

Example 6 — Implementing Full Referential Integrity

This example shows how to enforce referential integrity on store, modify, and erase operations for the CITIES and TOURISM relations. In this example:

- The store trigger on the TOURISM relation ensures that the city name and state code on the new TOURISM record are valid before it allows the store to take place.
- The modify trigger on the TOURISM relation checks to see if a user is modifying the city name. If this is the case, the trigger ensures that the new city name is valid before it allows the modify to take place.

- The first modify trigger on the CITIES relation checks to see if a user is modifying the city name or state code of a state capital. If this is the case, the operation is aborted.
- The second modify trigger on the CITIES relation is activated only if the modify of the CITIES relation takes place. It cascades the modify to the CITY field in the TOURISM relation.
- The erase trigger on the CITIES relation checks to see if a user is erasing a state capital. If this is the case, the operation is aborted.

  If the city being erased is not a state capital, the trigger then deletes all records in the TOURISM relation associated with that city.

```
/* Pre store trigger for TOURISM */
define trigger valid_store for tourism
pre store 0:
begin
  if not any c in cities with c.city = new.city
    and c.state = new.state
    abort 1;
end;
end_trigger
message 1: "Invalid city.";

/* Pre modify trigger for TOURISM  */
define trigger valid_modify for tourism
pre modify 0:
begin
if (new.city ne old.city)
  or (new.state ne old.state)
begin
  if not any c in cities with c.city = new.city
    and c.state = new.state
  abort 1;
end;
end;
end_trigger
message 1: "Invalid city.";

/* Pre modify trigger for CITIES */
define trigger retain_capitals for cities
pre modify 0:
begin
  if (new.city ne old.city)
    or (new.state ne old.state)
```

```
  begin
    for st in states with st.capital = old.city
      and st.state = old.state
    abort 1;
    end_for;
  end;
end;
end_trigger
message 1: "You can't modify a state capital.";
/* Post modify trigger for CITIES */
define trigger cascading_modify for cities
post modify 0:
begin
  if (new.city ne old.city)
    or (new.state ne old.state)
  begin
    for t in tourism with t.city = old.city
      and t.state = old.state
    modify t using
      t.city = new.city,
      t.state = new.state;
    end_modify;
    end_for;
  end;
end;
end_trigger;

/* Pre erase trigger for CITIES */
define trigger cascading_erase for cities
pre erase 0:
begin
  for st in states with st.capital = old.city
    and st.state = old.state
    abort 99;
    end_for;
  for t in tourism with t.city = old.city
    and t.state = old.state
    erase t;
  end_for;
end;
end_trigger
message 99: "You can't erase a state capital.";
```

For More Information

For more information on using triggers, refer to the following entries in the *DDL Reference:*

- **define trigger**
- **modify trigger**
- **delete**

For more information on defining unique indexes, refer to Chapter 6, *Defining Views and Indexes*, and to the following entries in the *DDL Reference:*

- **define index**
- **modify index**
- **delete**

For more information on defining validation criteria for fields, refer to Chapter 4, *Defining Fields,* and to the following entries in the *DDL Reference:*

- **define field**
- **modify field**
- **delete**

Chapter 8
Securing Data and Metadata

This chapter describes how to secure data and metadata by using InterBase security classes. It presents an overview of the security class mechanism and then describes how to define a security class, assign a security class to an object, design a security scheme, and change security definitions.

InterBase also allows you to secure specific relations from unauthorized access by using the SQL **grant** and **revoke** statements. This security scheme is discussed in the chapter on defining metadata with SQL in the *Programmer's Guide*.

Overview

By default, data and metadata on an InterBase database are not secured. When you first create a database, users have unlimited access to its relations, views, and fields.

InterBase provides a security scheme that is based on the concept of an *object*. An object is a database component that can be protected from individual users and groups.

The following database components are objects:

- The database itself
- Relations
- Views
- Fields in a relation or view

InterBase security operates within the context of file-level security. Users must have write access to the database file in order to access the data in the database.

Securing an Object

You secure an object from unauthorized access by assigning it a *security class.* This automatically prevents a user from accessing the object unless the user has been authorized to use that security class.

Security classes protect an object from unauthorized access through any interface, including **gdef**, **qli**, GDML, and SQL. They also prevent unauthorized users from accessing the object's definition through system relations.

Note

You can't assign a security class to relations created with the SQL **create table** command. Instead, you control access to these relations by using the SQL **grant** and **revoke** statements.

Granting Access Privileges

You control what *kind* of access the user has to an object by granting that user certain access privileges. Table 8-1 describes the privileges you can grant for a particular object.

Table 8-1. Access Privileges

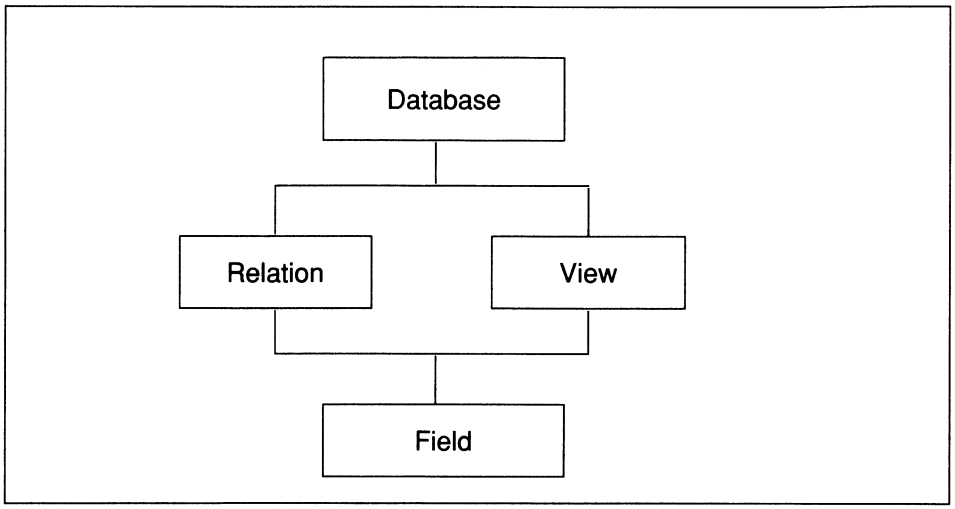
Privilege	What users can do
Read (r)	Read data from the object.
Write (w)	Write data to the object and delete the object.
Delete (d)	Delete the object's definition.
Control (c)	Change, but not delete the object's definition.
Protect (p)	Change the object's security class

The InterBase Security Hierarchy

InterBase security operates in a hierarchical fashion, with the database being the highest level object and the field the lowest. Higher-level access controls always override lower-level ones.

Figure 8-1 illustrates this hierarchy.

Figure 8-1. The InterBase Security Hierarchy



For example, users can't store data in a relation unless they have write privileges for the:

- Database that contains the relation
- Relation itself

Defining a Security Class

You define a security class by using the **define security_class** statement. You use this statement to name the security class and to define the access of indvidual users, groups, and views. This definition is called an *access control list* or an *ACL*.

When you define a security class, be sure you give at least one user full access privileges (rwdcp). If you don't do this, you can lock access to associated objects. This is especially important if you plan to associate the security class with a database.

If You Lock Yourself Out

If you do lock yourself out of a database, here's what to do:

- On Apollo systems, login as *locksmith* and fix the problem.
- On UNIX systems, login as *superuser* and fix the problem.
- On VMS systems, use the *bypass* privilege to override security and fix the problem.

Required Authority

To define a security class, you must have write privileges for the system relation RDB$-SECURITY_CLASSES. This relation is described in *Appendix A, System Relations*.

Example of Defining a Security Class

Suppose you want to secure a personnel database containing confidential salary information in the EMPLOYEE relation. You want to grant the following types of access:

1. The Vice President of Personnel, user ppn, should have complete access to salary information, including the authority to secure this information and to change its metadata.
2. Other Personnel employees should have read access to salary information.
3. No one else should have access to salary information.

There are two ways you can implement this security scheme.

Method 1

One way to implement this scheme is to define a security class that gives access privileges to the appropriate individuals and groups. For example, to define the security class on an Apollo system, type:

```
modify database 'personnel.gdb'
define security_class salary_access
    {This security class is used to secure salary data from
     unauthorized access}
    ppn.personnel rwdcp,
    %.personnel r;
```

To define this security class under UNIX and VMS, type:

```
modify database 'personnel.gdb'
define security_class salary_access
    {This security class is used to secure salary data from
     unauthorized access}
    [201,114] rwdcp,
    [201,*] r;
```

Later on you would assign this security class to the SALARY field in the personnel database, as shown later in this chapter.

Note

When you define the security class under UNIX and VMS, you must enclose the group and user codes in brackets. The syntax for the UNIX grantee for the **define security_class** statement is shown below. The "groupid" and "userid" must be numbers.

```
UNIX: grantee:==[groupid,userid]
```

Method 2

Another way to implement this scheme is to define a security class that gives full access privileges to user ppn and read privileges to a view.

For example, to define this security class on an Apollo system, type:

```
modify database 'personnel.gdb'
define security_class salary_access
    {This security class is used to secure salary data from
     unauthorized access}
    ppn.personnel rwdcp,
    view payroll r;
```

To define this security class under UNIX and VMS, type:

```
modify database 'personnel.gdb'
define security_class salary_access
    {This security class is used to secure salary data from
     unauthorized access}
    [201,114] rwdcp,
    view payroll r;
```

Later on, you would create a PAYROLL view of the EMPLOYEE relation and give payroll employees read access to the view. These employees would then have read access to the SALARY field.

The use of views in a security scheme is discussed later in this chapter.

Considerations for Defining Security Classes

The following considerations apply to defining security classes:

- You should always define your ACLs by using the conventions of the system on which the database resides. For example, you should use VMS user identification conventions when you define a database that resides on a VMS system.

  Note

  The security examples in the remainder of this chapter use Apollo user identification conventions.

- When you access a database from a node on a different system, InterBase automatically translates the ACLs on that database to ACLs that your system understands.
- When you use **gdef extract** to copy a database definition to a node on a different system, or when you use **gbak** to back up and restore your database to a node on a different system, you must change the ACLs. If you don't change the ACLs, **gdef** could fail.

Assigning a Security Class to an Object

You assign a security class to an object by including the security class in the object's definition. For example, to assign the salary_access security class to the SALARY field in the personnel database, type:

```
modify database 'personnel.gdb';
define field salary
   {used to hold salary information}
   security_class salary_access ....;
```

To assign a security class to an object, you must have protect privileges for that object.

Note

You can assign only one security class to a particular object.

Designing a Security Scheme

The InterBase security system gives you the capability of making your security scheme as granular as necessary. You can:

- Define as many security classes as you like.
- Associate one security class with any number of objects.
- Include any number of access definitions within one security class.

You can also achieve varying degrees of granularity by using views in your security scheme.

Note

Users can change security class definitions by manipulating the RDB$SECURITY_CLASSES systems relation. To prevent this capability, *be sure to secure this relation.*

This subsection tells you how to order your access definitions when you design a security scheme. This is followed by a discussion of views and examples of various security implementations.

Ordering Your Access Definitions

InterBase evaluates access definitions in the order they appear. Because of this, you need to order these definitions from most specific to least specific. If you don't order definitions in this way, you might get unexpected results.

Note

You don't have to be concerned about ordering access definitions, if your database resides on an Apollo. In this case, InterBase automatically orders the definitions most specific to least specific, regardless of the sequence in which they are entered.

For example, suppose you want to secure the customer relation and provide the following accesses:

- The sales manager, user jkw, should have unlimited access to the relation.
- Each sales person should have read, write access to their own customer information. (This can be done through a view.)
- Each sales person should have read access to all customer information.

Now, suppose you assign this security class to the customer relation:

```
modify database 'customer.gdb';
define security_class customer_access
  {This security class is used to secure customer data from
  unauthorized access}
  %.%.sales_people r,
  view my_customer rw,
  jkw.manager.sales_people rwdcp;
```

With this definition, you probably will not get the results you want. When the sales manager tries to access the customer relation, the manager is granted read access only. This is because the manager's id matches the first access definition. The same holds true when an individual sales person tries to access the relation.

The correct security class definition would look like this:

```
modify database 'customer.gdb';
define security_class customer_access
  {This security class is used to secure customer data from
  unauthorized access}
  jkw.manager.sales_people rwdcp,
  view my_customer rw,
  %.%.sales_people r;
```

Using Views

You can use views to grant a user or group access to an object. If a user has access to a view, that user may be able to access one or more base relations through the view. The same holds true for a group.

Granting access through a view works as follows:

- *If the view is not secured*, all users can access the base relations through the view. Users are automatically granted the same privileges that the view has for the base relations.

  For example, the security class LIMITED_ACCESS gives read access of the EMPLOYEE relation to the view UNDERPAID_EMPLOYEES. Since UNDERPAID_EMPLOYEES is not secured, all users can use this view to read certain fields in the EMPLOYEE relation:

```
define security_class limited_access
  ppn.personel rwdcp,
  view underpaid_employees r;

modify relation employee
```

```
        security_class limited_access;

define view underpaid_employees
   of emp in employee with emp.salary < 2000
   emp.emp_id,
   emp.last_name,
   emp.first_name;
```

- *If the view is secured*, only users authorized to access the view can use it to access the base relation. These users are granted whatever privileges form the intersection between the user's access to the view and the view's access to the base relation.

  The following diagram and table illustrate this concept.

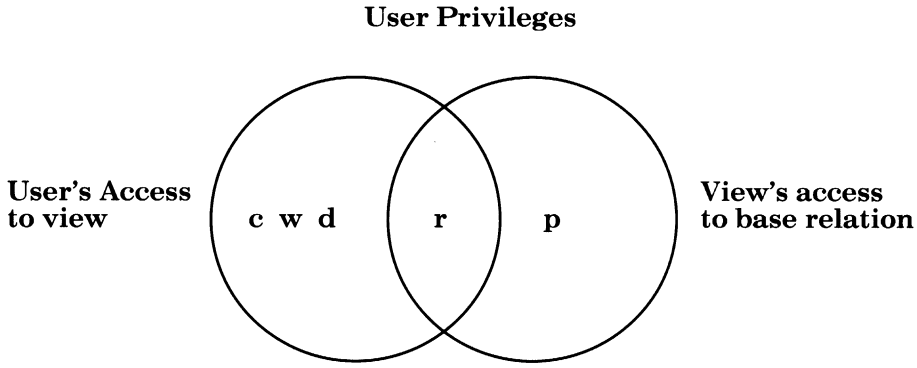


Table 8-2. Using a View to Access a Relation

User's Access to View	View's Access to Base Relation	What the User Can Do
rw	rw	Read and write to the base relation
rw	r	Read the base relation
r	rw	Read the base relation

Examples

The examples below illustrate various security implementations.

Example 1

Consider the following scenario for an organization with a highly stratified personnel policy:

1. The vice president of personnel can read and update any personnel information.
2. Other employees can know only the salary scales of jobs within two job levels of their own.
3. No one can know anyone's salary except his or her own.
4. Payroll clerks must be able to enter all information about every job and every employee, except the employee's review.
5. Only the personnel vice president can update reviews.
6. Any employee can read his or her own review.
7. Anybody can get a list of employee names.

The following database definition follows all seven rules. If you try the example, remember to add your own user identification, giving yourself the privileges of the personnel vice president.

Note

The database includes a field USER_NAME that is used in conjunction with a field called RDB$USER_NAME. You can use this expression in triggers, views, and validation.

```
define database "emp.gdb";

define field last_name varying [20];
define field first_name varying [10];
define field user_name varying [20];
define field salary long scale -2;
define field review blob segment_length 80;
define field job varying [20];
define field job_code short;

define security_class exclusive
  { for protecting rdb$security_classes
  and reviews }
ppn.vice_president.personnel rwdcp,
view censored_employees r,
```

```
define security_class personnel
  { to give reasonable access to reasonable
  people, if they ask for the right things }
ppn.vice_president.personnel rwdcp,
view censored_employees r,
view emp_names r,
clerk.payroll.personnel w;

define relation jobs
  security_class personnel
  { list of job names, job codes, and salary
  ranges for every job in the company.}
job,
job_code,
low_salary based on salary,
high_salary based on salary;

define relation employees
  security_class personnel
  { Information in this relation would cause
  a riot if the other vice presidents saw
  it, and a local massacre if the president
  finds out... }
last_name,
first_name,
user_name,
salary,
job_code;

define relation reviews
  security_class exclusive
  { Very hush hush.  }
last_name,
first_name,
review;

define view censored_employees of
  e in employees cross r in reviews over
  last_name, first_name with e.user_name = rdb$user_name
  { Each person can look at his own
  information but no one else's.}
e.last_name,
e.first_name,
e.user_name,
e.salary,
```

```
e.job_code,
r.review;

define view emp_names of e in employees
  { Everybody can get a list of employees }
name computed by (e.last_name | ', ' | e.first_name);

define view censored_jobs of
  j in jobs cross e in employees with
  e.user_name = rdb$user_name and
  j.job_code between (e.job_code - 2) and
  (e.job_code + 2)
  { Each person can see the salary ranges for
  jobs within two levels of his own }
j.job,
j.job_code,
j.low_salary,
j.high_salary;

modify relation rdb$security_classes security_class exclusive;
```

Example 2

This security scheme provides the following access to the SALARY field in the EMPLOYEES relation:

- The vice president of personnel has complete access to the salary field.
- Personnel managers have read and write access to the SALARY field.
- Members of the payroll department have write access to the SALARY field.

The other fields in the EMPLOYEES relation are not protected.

```
define security_class write_no_read
  { more access as you move up the corporate ladder }
     %.vp.personnel rwdcp,
     %.management.personnel, rw
     %.payroll.personnel; w

define relation employees
  last_name,
  first_name,
  emp_number,
  salary long
     security_class write_no_read;
```

Changing Your Security Scheme

Once you have implemented a security scheme you can easily change it if necessary. Changing a security scheme involves modifying security class definitions and assignments. Instructions for doing this are presented below.

Modifying a Security Class Definition

You modify a security class by deleting the security class and redefining it. For example, to modify the security class SALARY_ACCESS so that all personnel employees have at least read access to salary information, type:

```
modify database 'personnel.gdb'
delete security_class salary_access;

define security_class salary_access
{This security class is used to secure salary data from
     unauthorized access}
    ppn.vp.personnel rwdcp,
    %.payroll.personnel rw,
    %.%.personnel r;
```

Modifying a Security Class Assignment

You modify a security class assignment by using the appropriate **modify** statement to delete a security class from a database, relation, field, or view definition.

For example, to delete the association between the security class SALARY_ACCESS and the salary field, type:

```
modify database 'personnel.gdb';

modify field salary
drop security_class;
```

For More Information

For more information on security classes, refer to the discussion of the following statements in the *DDL Reference.*

- **define database**
- **define field**
- **define relation**
- **define security_class**
- **define view**
- **delete**
- **modify database**
- **modify field**
- **modify relation**
- **modify view**

For information on SQL security, refer to the chapter on defining metadata with SQL in the *Programmer's Guide.*

Chapter 9
Creating User-Defined Functions

This chapter discusses how to code, define, and access user-defined functions. It also discusses how to create a function library for storing the functions.

Overview

User-defined functions are executable routines that you define to the database. A user-defined function is a function in the strict theoretical sense. It takes zero or more arguments and returns a single value.

You can access user-defined functions either through **qli** or through a host-language program that contains embedded GDML statements. You can also access user-defined functions through **gdef** and include them in computed field definitions and trigger definitions.

You can create user-defined functions to do any number of conversion and calculations. For example, you can create a function that changes the case of text from lower case to uppercase. Or, you can create a function that calculates the absolute value of a given input value.

To create and use user-defined functions, follow these steps:

1. Write the functions and compile them into object code.
2. Define the functions to the database.
3. Create the function library and make it available to InterBase at run time.
4. Access the functions from **qli** or a host-language program.

Each of these functions are described in the following sections.

Writing and Compiling Functions

You can write a user-defined function by using any host language callable from C. The example below shows how to write four user-defined functions in C:

- The first function returns the absolute value of a number passed as an input argument.
- The second function takes a text string passed as an input argument and converts it to upper case. It then returns the converted string to the calling program.
- The third function takes two numbers as input arguments and returns the number that's of greater value.
- The fourth function returns the current time to the calling program. This function does not expect an input argument.

These functions are coded in a single file called *udf.c*. For additional examples of functions, see the InterBase examples directory.

```
#include <math.h>
#include <ctype.h>
#include <time.h>

/* variable to return values which is global so it stays
   around after the function invocation exits */

static char buffer[256];
static double retval;

/*============================================================
fn_abs() - returns the absolute value of its argument.
       define function abs
       module_name 'FUNCLIB'
       entry_point 'FN_ABS'
       double by value,
       double by value return_value
============================================================*/
double *fn_abs(x)
    double *x;
{
retval = fabs(*x);
return retval;
}

/* fn_upper() - Converts a null-terminated string to upper case
*/
```

```
char *fn_upper (s)
    char *s;
{
char *buf;

for (buf = buffer; *s;)
    if (*s >= 'a' && *s <= 'z')
        *buf++ = toupper(*s++);
    else
   *buf++ = *s++;

*buf = '\0';

return buffer;
}

/* fn_max() - Returns the greater of its two arguments */

double fn_max(a,b)
    double *a, *b;
{
return  (*a > *b) ? *a : *b;
}

/* fn_time() - Returns the current time */

char *fn_time()
{
int i, time_int;
char *buf, *end, *time_str;

strcpy (buffer, "The time is now ");
buf = buffer + strlen(buffer);

time (&time_int);
time_str = ctime (&time_int) + 11;

for (i = 0; i < 8; i++)
   *buf++ = *time_str++;

for (end = buffer + sizeof (buffer); buf < end;)
    *buf++ = ' ';
```

```
return buffer;
}
```

You compile these functions as follows:

- Under Apollo SR9.7, type:

  ```
  cc -c udf.c
  ```

- Under Apollo SR10.0, type:

  ```
  cc -c -W0,-pic udf.c
  ```

- Under SunOS 4.0, type the following for inclusion in a shareable object library:

  ```
  cc -c -pic udf.c
  ```

  Type the following for inclusion in a non-shareable object library:

  ```
  cc -c udf.c
  ```

- Under UNIX systems that don't support dynamic libraries, type:

  ```
  cc -c udf.c
  ```

- Under VMS, type:

  ```
  cc/gfloat udf.c
  ```

These commands create an object module called *udf.o*, except under VMS, where the output file is called *udf.obj*.

Defining Functions to the Database

You define a function to the database by specifying the following information:

- Function name.
- Name of the library where the function is stored.
- Entry point in the code.
- Datatype of each argument that gets passed to the function. You can specify any valid datatype except for the array datatype. You can also specify a cstring datatype to represent arguments that are null terminated strings.

  If you use this datatype, your program can pass character data to the function as a null terminated string. If you don't use this datatype, your program has to pad the full length of the string with spaces.

  The cstring datatype is valid for user-defined functions only. It is not valid in a field definition.
- The argument is passed to the function as a pointer (by reference). This is determined by the function code and the calling convention of the language being used. If the function expects to see pointer, pass the input argument by reference.

  Some languages, like Pascal, always expect their arguments to be passed by reference, no matter how the argument is declared in the function. For more information on passing arguments, refer to the document for your operating system that discusses conventions for calling other languages from C .

  You shouldn't attempt to modify an argument in a user-defined function, even if the argument was passed by reference. The **reference** option enables users to inform the database of their language's calling conventions. It was not designed to give users read/write arguments.
- An indicator to mark a single argument as the return argument.
- The datatype of the return argument. You can specify any valid datatype except for the array datatype. You can also specify a cstring datatype to represent an argument that is a null terminated string.

  If you use this datatype, the function can pass character data to the program as a null terminated string. If you don't use this datatype, the function has to pad the full length of the string with spaces.

  The cstring datatype is valid for user-defined functions only. It is not valid in a field definition.
- The argument is passed to the calling program as a pointer. This is also determined by the function code and the calling convention of the language being used.

- A query name that can be used when the function is accessed through **qli**. This information is optional. If you don't specify a query name, you can access the function by using its function name.
- User defined functions can be used in a **valid_if** clause.

When you define a function to the database, you can specify as many input arguments as you need. You must specify exactly one return argument. You designate an argument as a return argument by attaching the **return_value** option to the argument specification.

The definitions for the functions coded earlier in this chapter are shown below:

```
modify database 'atlas.gdb';

define function ABS
  module_name 'FUNCLIB'
  entry_point 'FN_ABS'
  double by reference,
  double by value return_value;

define function UPPER
  module_name 'FUNCLIB'
  entry_point 'FN_UPPER'
  cstring[256] by reference return_value,
  cstring[256] by reference;

define function MAXNUM
  module_name 'FUNCLIB'
  entry_point 'FN_MAX'
  double by reference,
  double by reference,
  double by value return_value;

define function TIME
  module_name 'FUNCLIB'
  entry_point 'FN_TIME'
  char[35] by reference return_value;
```

Notes

For Apollo SR9.7 systems, you need to uppercase the library name specified in the function definition. This ensures that the name matches the **bind** output.

For Apollo SR10.0 systems, you need to lowercase the entry point specified in the function definition.

If you need to delete one of the functions you just defined, use the **delete** statement:

```
delete function ABS;
```

If you need to modify one of the functions you just defined, delete the function and then redefine it:

```
delete function ABS;
define function ABS
   module_name 'SAMPLIB'
   entry_point 'FN_ABS'
   double by reference,
   double by value return_value;
```

To view the definition of a function, use the **qli show function** statement. This statement is described in the *QLI Reference.*

Creating a Function Library

You can create a function library on any platform that InterBase supports. You should create one function library for each platform on which the database involved resides.

To modify an existing function library function on all platforms:

- Compile the function according to the instructions for your platform.
- Include all object files previously included in the library in addition to the newly-created object file in the command line when creating the function library.

Note

On some platforms, it is possible to link object files directly with existing libraries. For more information, consult the documentation for your operating system.

Instructions for creating function libraries and making them available to InterBase on each supported platform are presented below.

Creating a Function Library Under Apollo

To create a function library under Apollo SR9.7 or SR10.0, use the **bind** utility to bind the object you just created into a function library. When you do this, you must include a **-mark** option for each function entry point.

For example, to bind *udf.o* into a library named *funclib*, type:

```
% bind udf.o -bin funclib -mark fn_abs -mark fn_upper -
      -mark fn_max -mark fn_time;
```

If you want to provide access to the function library locally, use the **inlib** command to make the library available to the process you are using:

```
% inlib funclib
```

If you want to access the function library through the remote server, install the function library as a global library on each node. This ensures the library will be available to all processes that need to access it:

- To do this under Apollo SR9.7, rebind the **gdslib** to **inlib** the function library:

```
% cd /interbase/lib
% cp gdslib gdslib.orig
% bind -inlib funclib -bin gdslib gdslib.orig
```

- To do this under Apollo SR10.0, add the function library to the */etc/sys.conf* file and perform a soft reboot of the system.

Creating a Function Library Under SunOS 4.0

To create a function library under SunOS 4.0 by using shareable libraries:

1. Log on to the node where the database resides.
2. Add the name, entry point, and module name of each function to the table ISC_FUNCTIONS in */usr/interbase/examples/functions.c*. Later on, when you define the function to the database, be sure to specify the module name and entry point exactly as they're specified here.

Note

The function name you enter in ISC_FUNCTIONS must be unique.

ISC_FUNCTIONS provides a template for you to follow. The first time you open the file, you'll see the following description:

```
typedef struct {
     char        *fn_module;
     char        *fn_entrypoint;
     FUN_PTR     fn_function;
}FN;

static test();

static FN      isc_functions [] = {
     "test_module", "test_function", test,
    0, 0, 0};
```

To fill in the ISC_FUNCTIONS table for the FN_ABS, FN_UPPER, FN_MAX, and FN_TIME functions, type:

```
extern double fn_abs();
extern char *fn_upper ();
extern double fn_max ();
extern char *fn_time ();

static FN      isc_functions [] = {
     "test_module", "test_function", test,
     "FUNCLIB", "FN_ABS", (FUN_PTR)fn_abs,
     "FUNCLIB", "FN_UPPER", (FUN_PTR)fn_upper,
     "FUNCLIB", "FN_MAX", (FUN_PTR)fn_max,
     "FUNCLIB", "FN_TIME", (FUN_PTR)fn_time,
    0, 0, 0};
```

Note

Don't delete the final line of zeroes. They signal the end of the table.

3. Compile *functions.c*:

```
% cc -c -pic functions.c
```

4. Copy the shareable library */usr/interbase/lib/gdsflib.so.0.0* to your working directory:

```
% cp /usr/interbase/lib/gdsflib.so.0.0 new_gdsflib
```

5. Add the object files *udf.o* and *functions.o* to the shareable library:

```
% ld -o new_gdsflib -assert pure-text udf.o functions.o
```

To make the function library available under SunOS 4.0 by using shareable libraries:

6. Set up an environment variable to force the use of the new shareable library for testing purposes:

```
% setenv LD_LIBRARY_PATH /absolute directory path of
      new_gdsflib
```

7. Link */usr/interbase/lib/gdslib.so.0.0* to your working directory:

```
% ln -s /usr/interbase/lib/gdslib.so.0.0 libgdslib.so.0.0
```

8. Link *new_gdsflib* to the filename *libgdsflib.so.0.0*. This enables InterBase to find *new_gdsflib* at run-time:

```
% ln -s new_gdsflib libgdsflib.so.0.0
```

9. Test your functions.
10. Copy the new shareable library, *new_gdsflib*, to */usr/interbase/lib* on the node where the database resides:

```
% cp new_gdsflib /usr/interbase/lib/gdsflib.so.0.0
```

To create a function library under SunOS 4.0 without using shareable libraries, follow the instructions below for creating the function library under other UNIX platforms.

Creating a Function Library Under Other UNIX platforms

To create a function library under SunOS 3.5, HP-UX, and Ultrix:

1. Log on to the node where the database resides.
2. Add the name, entry point, and module name of each function to the table ISC_FUNCTIONS in */usr/interbase/examples/functions.c.*

   For example, to fill in the ISC_FUNCTIONS table for the functions you coded earlier, type:

```
static FN       isc_functions [] = {
```

```
    "test_module", "test_function", test,
    "FUNCLIB", "FN_ABS", abs,
    "FUNCLIB", "FN_UPPER", upper,
    "FUNCLIB", "FN_MAX", maxnum,
    "FUNCLIB", "FN_TIME", time,
   0, 0, 0};
```

3. Compile *functions.c* without using the **pic** switch:

```
% cc -c functions.c
```

4. Concatenate *functions.o* with the file that contains the function code. In this example, the file is called *udf.o*:

```
% ld -r -o funclib.o functions.o udf.o
```

5. Copy the InterBase back end to your working directory:

```
% cp /usr/interbase/lib/gds_b.a new_gds_b.a
```

6. Remove the old *functions.o* from the INTERBASE back end using the **ar** utility when building user-defined functions:

```
% ar dv new_gds_b.a functions.o
```

7. Add the function library to the new back end:

```
% ar rls new_gds_b.a funclib.o
```

8. Reinitialize the symbol table for the archive:

```
% ranlib new_gds_b.a
```

To make the function library available under these operating systems, you need to build the function library into the pipe server by following these steps:

1. Link the InterBase back end to the pipe server:

```
% cc /usr/interbase/lib/gds_pipe.a new_gds_b.a -o gds_pipe
```

   Depending on your platform and the type of functions you have defined, you may also need to link with the math library provided with your platform.

2. Set up an environment variable to force the use of the new pipe server for testing purposes:

```
% setenv GDS_SERVER /absolute directory path of your
      working directory/gds_pipe
```

3. Test your functions by using **qli** or a GDML application program. Information on accessing functions is presented later in this chapter.
4. When you are satisfied with how your functions work, save the original version of the pipe server and the original version of *gds_b.a*:

```
% cp /usr/interbase/bin/gds_pipe
   /usr/interbase/bin/gds_pipe.bak

% cp /usr/interbase/lib/gds_b.a /usr/interbase/lib/gds_b.bak
```

5. Replace the original version of the pipe server with the pipe server you created in Step 1:

```
% cp gds_pipe /usr/interbase/bin/gds_pipe
```

6. Copy the new InterBase back end to */usr/interbase/lib*:

```
% cp  new_gds_b.a /usr/interbase/lib/gds_b.a
```

7. Relink any applications that were previously linked against *gds_b.a*.

If you want to be able to access your functions remotely, you must also rebuild the inet server so that it incorporates the new functions. To do this, follow these steps:

1. Link *gds_inet_server* to the pipe server:

```
% cc /usr/interbase/lib/gds_inet_server.a
      new_gds_b.a -o new_gds_inet_server
```

2. Test your functions remotely by using **qli** or a GDML application program. Information on accessing functions is presented later in this chapter.
3. When you are satisfied with how your functions work, save the original version of the inet server:

```
% cp /usr/interbase/bin/gds_inet_server.a
      /usr/interbase/bin/gds_inet_server.bak
```

4. Replace the original version of the inet server with the inet server you created in Step 1:

```
% cp new_gds_inet_server /usr/interbase/bin/gds_inet_server
```

Creating a Function Library Under VMS

To create a function library under VMS, use a linker options file to make the function library entry points universal:

```
link/share=funclib.exe udf,sys$input/opt
  psect_attr = errno, noshr
  psect_attr = stderr, noshr
  psect_attr = stdin, noshr
  psect_attr = stdout, noshr
  psect_attr = sys_nerr, noshr
  psect_attr = vaxc$errno, noshr
```

```
universal = fn_abs
universal = fn_upper
universal = fn_max
universal = fn_time
```

By not using shared writeable psects, you avoid having to use the **install** utility to install the shared function library. For more information on shareable executables, refer to the VAX Linker documentation.

To make the function library available under VMS, do any of the following:

- Copy the shareable executable to the *sys$share* directory:

  ```
  $ copy funclib.exe sys$share:
  ```

- Make *sys$share* a list of pathnames that includes the directory containing the function library:

  ```
  $ define sys$share $mydisk:[mydir], sys$sysroot:[syslib]
  ```

- Define a logical name that has the same name as the function library module:

  ```
  $ define funclib $mydisk:[mydir]funclib.exe
  ```

Accessing Functions

Instructions for accessing user-defined functions through **qli** and host-language programs are presented below.

Accessing Functions From Qli

To access a user-defined function from **qli**, name the function, and enclose the input arguments in parentheses.

For example, to access the ABS function from **qli**, type:

```
QLI> ready atlas.gdb;
QLI> print abs(-3);
 3

QLI> declare foo double;
QLI> foo = -3;
QLI> print abs(foo);
 3
```

To access the MAXNUM function from **qli**, type:

```
QLI> ready atlas.gdb;
QLI> print maxnum (2, 5);
 5
```

Accessing Functions From a Host-Language Program

To access a function from a host-language program, name the function and enclose the input arguments in parentheses. The function can only be accessed from a GDML record selection expression.

The example below shows how to access the ABS function from a C program:

```
database atlas = filename "atlas.gdb"

main()
{

ready atlas;
start_transaction;

printf("This program prints the states that have changed in
  population\n");
```

```
printf("by more than 250,000 people between 1970 and
  1980.\n\n");

printf("State          census_1970          census_1980\n\n");
  for p in populations with
       abs(p.census_1970 - p.census_1980) > 250000 sorted by
            descending abs(p.census_1970 - p.census_1980)
        printf("%4s          %10d           %10d\n", p.state,
            p.census_1970, p.census_1980);
end_for;
rollback;
finish;
}
```

Accessing the Functions From Gdef

You can access a user-defined function in a computed field definition and a trigger definition.

For example, the statement below uses the ABS function to compute the absolute value of the difference between the 1970 population and the 1980 population:

```
define view pop_change of p in populations
  change computed by
  (abs(p.census_1970 - p.census_1980));
```

This statement uses the UPPER function to convert new state names to upper case:

```
define trigger up_name for states
pre store 0:
  new.state = upper(new.state);
end_trigger;
```

For More Information

For more information on writing user-defined functions and accessing them from a host-language program, refer to the chapter on retrieving data with GDML in the *Programmer's Guide*.

Chapter 10
Creating Event Alerters

This chapter introduces the InterBase event alerter mechanism, describes how this mechanism works, and discusses event transaction control. For an in-depth discussion of programming with events, refer to the chapter on programming with events in the *Programmer's Guide*.

Overview

An *event alerter* is a mechanism that notifies an interested application when a specific event has taken place. An *event* can be any type of database insertion, modification, or deletion.

For example, suppose an investment bank wants to track changes to the stocks in its portfolio. The bank has an application program that inputs stock prices from the ticker and updates the price of each stock in the database. The bank now needs another program that determines whether to buy or sell stock based on changes to the stock price.

You can use the event alerter mechanism to notify the program when specified changes take place. The program can then act on these changes appropriately.

What Happens on the Database Side

To give this program timely notification of meaningful stock price changes, you can define an event in the STOCKS relation of the database. This event keeps track of stock changes and posts to all interested programs when the price change exceeds 1%.

You define and post an event in the database by using a trigger. The trigger must do the following:

- Identify the event by specifying a unique string
- Specify the conditions under which the event manager will notify interested programs that the event has occurred

For example, to post an event that monitors a 1% change in the price of stock, type:

```
define trigger stock_event for stocks
  post modify 0:
      if new.price / old.price > 1.01 or
         new.price / old.price < .99
      then post new.company;
end_trigger;
```

This trigger checks to see if the change in stock prices exceeds 1%. When the change does exceed 1%, the trigger posts the event. This consists of passing the name contained in its argument to the Interbase *event manager*. The event manager then checks to see if the name is in the *event table*, which lists the events in which active programs have registered interest.

Triggers that post events can differ from this example in a number of ways:

- You can use as much or as little conditional logic in the trigger as you need to qualify the event. However, a trigger that contains no conditional logic puts the burden of qualifying the event back onto the program. Avoid this type of inefficiency.
- The trigger can also contain several conditions, each of which, if satisfied, can post a different event.
- You can define the trigger with any valid characteristics, not just the defaults used here. You can also make it fire on other than a **modify**. For more information on defining triggers, refer to Chapter 7, *Preserving Data Integrity*.
- The argument to the **post** verb can be any alphanumeric name up to 31 characters in length. The event manager looks for an event that has same name.

What Happens on the Program Side

Programs can wait on many events and can choose whether to wait on them synchronously or asynchronously:

- If a program chooses to wait synchronously, the program gives control to InterBase. Control gets returned when an event is posted.

  This type of wait is supported through two GDML statements:

  — EVENT_INIT declares the names of events in which the program is interested.

  — EVENT_WAIT does the actual waiting, returning control when one of the declared events occurs.

- If a program chooses to wait asynchronously, the program keeps control while waiting. This enables the program to do other processing.

  This type of wait is supported only through InterBase access method calls.

Once a program is notified that an event has taken place, the program can check the **gds_$events** array to see which event it was.

The example below shows an embedded GDML program that waits synchronously for the STOCK_EVENT event to be posted. When this event occurs, the program prints the value of the changed stocks:

```
#include "/interbase/include/gds.ins.c"

DATABASE DB = "stocks.gdb";

#define number_of_stocks 5
char *event_names [] = { "APOLLO", "DEC", "HP", "IBM", "SUN" };

main() {

  int i;

  READY DB;

  EVENT_INIT PRICE_CHANGE ( "APOLLO", "DEC", "HP", "IBM", "SUN"
);

  while (1) {

      EVENT_WAIT PRICE_CHANGE;
```

```
        printf ("Get new event!\n");
        for (i=0;i<number_of_stocks;i++)
            printf ("Event status for company
                %s = %d\n",event_names[i], gds_$events[i]);

        for (i=0;i<number_of_stocks;i++) {
            if (gds_$events[i]) {
                START_TRANSACTION;
                FOR S IN STOCKS WITH S.COMPANY = event_names[i]
                    printf ("COMPANY: %s changed!  NEW PRICE:
%f\n",
                     S.COMPANY, S.PRICE);
                END_FOR;
                ROLLBACK;
                }
            }
        }
}
```

Transaction Control of Events

Events alerters are under transaction control, which means the program that sets the event trigger off can commit or roll back the event. Interested programs receive notification of an event only when the transaction completes.

An event can happen only once per transaction. Regardless of how many times a particular event is posted during a transaction, it's regarded as a single event for notification purposes.

For More Information

For more information on defining an event trigger, refer to the **define trigger** statement in the *DDL Reference*.

For more information on programming with events, refer to the chapter on programming with events in the *Programmer's Guide*.

Chapter 11
Modifying Metadata with Dynamic DDL

This chapter describes how to modify metadata with dynamic DDL. This involves generating a file of dynamic DDL (DYN) commands and executing those DYN commands from a host-language program.

Overview

Dynamic DDL is a mechanism you can use to modify metadata at runtime from third-generation language (3GL) programs.With this mechanism, **gdef** processes data definition source files and produces a data definition file that can be included in a 3GL program. You should consider using dynamic DDL to update databases when the use of **gdef** is inconvenient.

By using dynamic DDL, you can extend database definitions to support new features and then send users a program that updates their existing databases.

This **gdef** mechanism consists of three elements:

- A language called DYN that specifies metadata updates. **Gdef** is the only means for generating DYN commands.
- The **dynamic** option, which directs **gdef** to generate DYN commands.
- A **gds** routine called **gds_$ddl** that executes the DYN commands.

Note

Ada has special variations that are described in the Ada examples later in this chapter. If you program in Ada, be sure to read the variations described in these examples.

Generating and Using DYN Commands

To generate and use DYN commands:

1. Create a DDL source file to modify the database.
2. Back up your database.
3. Compile the source file using the **gdef dynamic** and **language** options.
4. Include the resulting data file in a program that readies the database, calls the **gds_$ddl** routine, commits its changes, and finishes the database.
5. Precompile the program with **gpre**, and then compile and link the program.
6. Run the program to modify a runtime version of the database.

These steps are described below and illustrated with an extended example that shows how to use DYN commands in a C language program. Examples with other programming languages are presented later in this chapter.

Creating the DDL Source File

Use an editor to create a source DDL file that contains the metadata changes you want to make.

In the extended example, the *football.gdl* source file adds the relation FOOTBALL_TEAMS to *atlas.gdb*. The new relation uses existing global fields and defines a new field called DOMED:

```
modify database 'atlas.gdb';

define relation football_teams
  team_name,
  city,
  state,
  home_stadium,
  seating,
  surface,
  domed char[1],
  league;
```

Backing Up Your Database

Before you compile the DDL source file, be sure to back up your database. For example, to back up your datababase in UNIX, type:

```
% gbak atlas.gdb atlas.gbak
```

When **gdef** processes the DDL file, it makes the changes directly to the database. You can't make those changes to the database again without generating errors. To avoid errors, you must run the application program on a copy of the database as it originally existed.

Compiling the DDL Source File

To compile the DDL source file, invoke **gdef** by using the following syntax:

Operating System	Syntax
Apollo AEGIS	% **gdef -dynamic** *dyn-filespec* [*-language*] *gdl-filespec*
UNIX	% **gdef -dynamic** *dyn-filespec [-language] gdl-filespec*
VMS	$ **gdef/dynamic** = *dyn-filespec* [*/language*] *gdl-filespec*

For example, to compile the *football.gdl* source file in UNIX, type:

```
% gdef -dynamic dyn.dat.c football.gdl
```

This writes the DYN commands to a readable data file called *dyn.dat.c*.

Note

The output file has exactly the same name given in the command line. No extensions are appended.

The dynamic DDL that **gdef** generates in this step is shown below:

```
gds_$dyn_version_1,
  gds_$dyn_begin,
      gds_$dyn_def_rel, 14,0,

'F','O','O','T','B','A','L','L','_','T','E','A','M','S',
          gds_$dyn_end,
      gds_$dyn_def_local_fld, 9,0,
              'T','E','A','M','_','N','A','M','E',
          gds_$dyn_rel_name, 14,0,

'F','O','O','T','B','A','L','L','_','T','E','A','M','S',
          gds_$dyn_fld_position, 2,0, 0,0,
          gds_$dyn_end,
      gds_$dyn_def_local_fld, 4,0,
              'C','I','T','Y',
          gds_$dyn_rel_name, 14,0,
```

```
'F','O','O','T','B','A','L','L','_','T','E','A','M','S',
         gds_$dyn_fld_position, 2,0, 1,0,
         gds_$dyn_end,
      gds_$dyn_def_local_fld, 5,0,
             'S','T','A','T','E',
         gds_$dyn_rel_name, 14,0,

'F','O','O','T','B','A','L','L','_','T','E','A','M','S',
         gds_$dyn_fld_position, 2,0, 2,0,
         gds_$dyn_end,
      gds_$dyn_def_local_fld, 12,0,
             'H','O','M','E','_','S','T','A','D','I','U','M',
         gds_$dyn_rel_name, 14,0,

'F','O','O','T','B','A','L','L','_','T','E','A','M','S',
         gds_$dyn_fld_position, 2,0, 3,0,
         gds_$dyn_end,
      gds_$dyn_def_local_fld, 7,0,
             'S','E','A','T','I','N','G',
         gds_$dyn_rel_name, 14,0,

'F','O','O','T','B','A','L','L','_','T','E','A','M','S',
         gds_$dyn_fld_position, 2,0, 4,0,
         gds_$dyn_end,
      gds_$dyn_def_local_fld, 7,0,
             'S','U','R','F','A','C','E',
         gds_$dyn_rel_name, 14,0,

'F','O','O','T','B','A','L','L','_','T','E','A','M','S',
         gds_$dyn_fld_position, 2,0, 5,0,
         gds_$dyn_end,
      gds_$dyn_def_global_fld, 5,0,
             'D','O','M','E','D',
         gds_$dyn_fld_type, 2,0, 14,0,
         gds_$dyn_fld_length, 2,0, 1,0,
         gds_$dyn_fld_scale, 2,0, 0,0,
         gds_$dyn_fld_sub_type, 2,0, 0,0,
         gds_$dyn_end,
      gds_$dyn_def_local_fld, 5,0,
             'D','O','M','E','D',
         gds_$dyn_rel_name, 14,0,

'F','O','O','T','B','A','L','L','_','T','E','A','M','S',
```

```
            gds_$dyn_fld_position, 2,0, 6,0,
            gds_$dyn_end,
        gds_$dyn_def_local_fld, 6,0,
                'L','E','A','G','U','E',
            gds_$dyn_rel_name, 14,0,

'F','O','O','T','B','A','L','L','_','T','E','A','M','S',
            gds_$dyn_fld_position, 2,0, 7,0,
            gds_$dyn_end,
        gds_$dyn_end,
gds_$dyn_eoc
```

Including the file in a program

Include the data file created by **gdef** in a program that calls **gds_$ddl**. This routine modifies data definitions by executing DYN commands.

The calling sequence of the **gds_$ddl** routine is shown below:

Syntax:

```
status = gds_$ddl (status_vector, db_handle,
transaction_handle, dyn_message_length,
dyn_message_address)
```

Arguments:

Parameter	Datatype	In/Out
status_vector	long	out
db_handle	ulong	inout
transaction_handle	ulong	inout
dyn_message_length	ushort	in
dyn_message_address	uspec	in

For more information about using **gds** calls in programs, see the chapter on using OSRI calls in the *Programmer's Guide.*

The C program *modify_atlas.e*, shown below, does the following:

- Includes the data file, *dyn.dat.c*
- Starts a transaction to modify the database
- Calls the **gds_$ddl** routine to execute the commands in the data file, and commits those changes

```
DATABASE DB = 'atlas.gdb';

static unsigned char dyn_gdl[] = {
#include "dyn.dat.c"
```

```
};

main ()
{
READY;
START_TRANSACTION;

gds_$ddl (gds_$status, &DB, &gds_$trans, sizeof (dyn_gdl),
dyn_gdl);

if (gds_$status[1])
  gds_$print_status (gds_$status);

COMMIT;
FINISH;
}
```

Precompiling, Compiling, and Linking the Program

Precompile the program by using **gpre**:

```
% gpre -n -m modify_atlas.e
```

Next, compile and link the program as you do with other DML programs.

The resulting program, *modify_atlas*, uses the metadata modifications in the data file included in the program and calls the access method to modify *atlas.gdb*. Runtime system users who don't have **gdef** can use such programs to modify data definitions.

For more information about precompiling programs with **gpre**, refer to the chapter on preprocessing your program in the *Programmer's Guide*.

Modifying the Data Definitions

Modify the data definitions by running the program that contains the DYN commands. You can verify the changes to the database by using **qli**:

```
% modify_atlas
% qli
Welcome to QLI
Query Language Interpreter
QLI> ready atlas.gdb
QLI> show relation football_teams
  FOOTBALL_TEAMS
```

```
    TEAM_NAME                   varying text, length 15
    CITY                        varying text, length 25
    STATE                       varying text, length 4
    HOME_STADIUM                varying text, length 30
    SEATING                     long binary
    SURFACE                     text, length 1
    DOMED                       text, length 1
    LEAGUE                      text, length 1
QLI> show field domed
Field DOMED in relation FOOTBALL_TEAMS of database QLI_0
    Global field DOMED
    Datatype information:
    text, length 1
```

Additional Examples

The following sections contain brief program examples for additional languages. The examples also include sample data files that result from **gdef**'s **dynamic** option, where applicable. In the Ada examples and the COBOL example, the output from the **dynamic** option is contained in the programs.

These brief programs modify a database named *example.gdb* using the data file and the **gds_$ddl** routine described in greater detail in previous sections. The programs modify *example.gdb* to add a relation called R that contains a field called I.

Apollo Ada Program Example

Ada programs can't call **gds** routines directly. Therefore this Ada example calls the *interbase.ddl* and *interbase.print_status* routines:

```
WITH basic_io, interbase;

PROCEDURE dyn_test IS

DATABASE DB = "example.gdb";

---  The following code, generated by gdef dynamic,
---  has been included here by the user.
gds_dyn_length: short_integer := 48;
gds_dyn: CONSTANT interbase.blr (1..48) := (
1,2,9,1,0,82,3,6,1,0,73,70,2,0,7,0,
71,2,0,2,0,72,2,0,0,0,73,2,0,0,0,3,
7,1,0,73,50,1,0,82,92,2,0,0,0,3,3,-1
);
---  End of included gdef code

begin

READY;
START_TRANSACTION;

interbase.ddl (gds_status, DB, gds_trans, gds_dyn_length,
gds_dyn'address);
if (gds_status (1) /= 0) then
  interbase.print_status (gds_status);
end if;
```

```
COMMIT;
FINISH;
end dyn_test;
```

Apollo FORTRAN Example

```
      PROGRAM TEST

%include 'dyn.dat.ftn.1'

      DATABASE DB = 'example.gdb'

%include 'dyn.dat.ftn.2'

      READY
      START_TRANSACTION

           call GDS_$DDL (gds_$status, DB, gds_$trans,
     +             gds_$dyn_length, gds_$dyn)
           if (gds_$status(2) .ne. 0) call gds_$print_status
    (gds_$status)

          COMMIT
          FINISH

          stop
          end
```

The data file *dyn.dat.ftn.1*:

```
           INTEGER*2 GDS_$DYN_LENGTH
           INTEGER*4 GDS_$DYN(12)
```

The data file *dyn.dat.ftn.2*:

```
           DATA GDS_$DYN_LENGTH  /48/
           DATA (GDS_$DYN(I) I=1,12)  /
          +
    16910593,5374726,16795974,33556224,1191313410,4719104,
          +  18690,3,117506121,838926418,1543634944,197631/
```

The output from **gdef -d** produces only one filename, which receives the name given on the command line. This file must be split into two files, as shown above. In the FORTRAN program code, the **include** statement for file "one" must be *above* the **db** statement; for file "two", below it.

Apollo Pascal Example

```
program test (input, output);

DATABASE DB = 'example.gdb'

var
%include 'dyn.dat.pas';

begin

READY;
START_TRANSACTION;

GDS_$DDL (gds_$status, DB, gds_$trans, gds_$dyn_length,
gds_$dyn);
if (gds_$status [2] <> 0) then
  gds_$print_status (gds_$status);

COMMIT;
FINISH;
end.
```

The data file, *dyn.dat.pas*:

```
gds_$dyn_length: integer16 := 48;
gds_$dyn: array [1..48] of char := [
    gds_$dyn_version_1,
       gds_$dyn_begin,
          gds_$dyn_def_rel, chr(1),chr(0), 'R',
             gds_$dyn_end,
          gds_$dyn_def_global_fld, chr(1),chr(0), 'I',
             gds_$dyn_fld_type, chr(2),chr(0), chr(7),chr(0),
             gds_$dyn_fld_length, chr(2),chr(0),
chr(2),chr(0),
             gds_$dyn_fld_scale, chr(2),chr(0),
chr(0),chr(0),
             gds_$dyn_fld_sub_type, chr(2),chr(0),
chr(0),chr(0),
             gds_$dyn_end,
          gds_$dyn_def_local_fld, chr(1),chr(0), 'I',
             gds_$dyn_rel_name, chr(1),chr(0), 'R',
             gds_$dyn_fld_position, chr(2),chr(0),
chr(0),chr(0),
```

```
                gds_$dyn_end,
            gds_$dyn_end,
        gds_$dyn_eoc
    ];  (* end of DYN string *)
```

VAX Ada Example

Ada programs can't call **gds** routines directly. Therefore this Ada example calls the *interbase.ddl* and *interbase.print_status* routines:

```
WITH basic_io, interbase;

PROCEDURE dyn_test IS

DATABASE DB = "example.gdb";

---  The following code, generated by gdef dynamic,
---  has been included here by the user.
gds_dyn_length: short_integer := 48;
gds_dyn: CONSTANT interbase.blr (1..48) := (
1,2,9,1,0,82,3,6,1,0,73,70,2,0,7,0,
71,2,0,2,0,72,2,0,0,0,73,2,0,0,0,3,
7,1,0,73,50,1,0,82,92,2,0,0,0,3,3,-1
);
---  End of included gdef code

begin

READY;
START_TRANSACTION;

interbase.ddl (gds_status, DB, gds_trans, gds_dyn_length,
gds_dyn'address);
if (gds_status (1) /= 0) then
  interbase.print_status (gds_status);
end if;

COMMIT;
FINISH;
end dyn_test;
```

VAX BASIC Example

```
10    %TITLE "TEST"

  DATABASE DB = 'example.gdb'

  %include 'dyn_dat.bas'

  READY
  START_TRANSACTION

  CALL GDS_$DDL BY REF (gds_$status, DB, gds_$trans, &
            gds_$dyn_length BY VALUE, gds_$dyn)
  if gds_$status(2) <> 0 then
      CALL gds_$print_status (gds_$status)
  end if

  COMMIT
  FINISH
  end
```

The data file, *dyn_dat.bas*:

```
  DECLARE WORD CONSTANT gds_$dyn_length = 48
  DECLARE STRING CONSTANT gds_$dyn =&
'1'C + '2'C + '9'C + '1'C + '0'C + 'R' + '3'C + '6'C + '1'C +  &
'0'C + 'I' + 'F' + '2'C + '0'C + '7'C + '0'C + 'G' + '2'C + '0'C
+  &
'2'C + '0'C + 'H' + '2'C + '0'C + '0'C + '0'C + 'I' + '2'C +  &
'0'C + '0'C + '0'C + '3'C + '7'C + '1'C + '0'C + 'I' + '2' +  &
'1'C + '0'C + 'R' + '' + '2'C + '0'C + '0'C + '0'C + '3'C +  &
'3'C + '255'C
```

VAX C Example

```
DATABASE DB = 'example.gdb';

static unsigned char dyn_gdl[] = {
#include "dyn_dat.c"
};

main()
{
READY;
```

```
START_TRANSACTION;

gds_$ddl (gds_$status, &DB, &gds_$trans, sizeof (dyn_gdl),
dyn_gdl);

if (gds_$status[1])
  gds_$print_status (gds_$status);

COMMIT;
FINISH;
}
```

The data file, *dyn_dat.c*:

```
gds_$dyn_version_1,
  gds_$dyn_begin,
      gds_$dyn_def_rel, 1,0, 'R',
          gds_$dyn_end,
      gds_$dyn_def_global_fld, 1,0, 'I',
          gds_$dyn_fld_type, 2,0, 7,0,
          gds_$dyn_fld_length, 2,0, 2,0,
          gds_$dyn_fld_scale, 2,0, 0,0,
          gds_$dyn_fld_sub_type, 2,0, 0,0,
          gds_$dyn_end,
      gds_$dyn_def_local_fld, 1,0, 'I',
          gds_$dyn_rel_name, 1,0, 'R',
          gds_$dyn_fld_position, 2,0, 0,0,
          gds_$dyn_end,
      gds_$dyn_end,
gds_$dyn_eoc
```

VAX COBOL Example

```
IDENTIFICATION DIVISION.
PROGRAM-ID. DYN_TEST.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
  DATABASE DB1 = FILENAME "example.gdb"
01 D-STAT PIC ZZZ9.
*  The following code, generated by gdef dynamic,
*  has been included here by the user.
01 GDS_$DYN_LENGTH PIC S9(4) USAGE COMP VALUE IS 48.
01 GDS_$DYN.
```

```
   03  GDS_$DYN_1 PIC S9(9) USAGE COMP VALUE IS 17367553.
   03  GDS_$DYN_2 PIC S9(9) USAGE COMP VALUE IS 100880896.
   03  GDS_$DYN_3 PIC S9(9) USAGE COMP VALUE IS 1179189249.
   03  GDS_$DYN_4 PIC S9(9) USAGE COMP VALUE IS 458754.
   03  GDS_$DYN_5 PIC S9(9) USAGE COMP VALUE IS 33555015.
   03  GDS_$DYN_6 PIC S9(9) USAGE COMP VALUE IS 149504.
   03  GDS_$DYN_7 PIC S9(9) USAGE COMP VALUE IS 38338560.
   03  GDS_$DYN_8 PIC S9(9) USAGE COMP VALUE IS 50331648.
   03  GDS_$DYN_9 PIC S9(9) USAGE COMP VALUE IS 1224737031.
   03  GDS_$DYN_10 PIC S9(9) USAGE COMP VALUE IS 1375732018.
   03  GDS_$DYN_11 PIC S9(9) USAGE COMP VALUE IS 604.
   03  GDS_$DYN_12 PIC S9(9) USAGE COMP VALUE IS -16579840.
*  End of included gdef code
PROCEDURE DIVISION.
MAIN-ROUTINE.


   READY.

   START_TRANSACTION.

   CALL "GDS_$DDL" USING GDS_$STATUS_VECTOR, BY REFERENCE DB1,
-        gds_$trans, BY VALUE GDS_$DYN_LENGTH,
-        BY REFERENCE GDS_$DYN
   IF GDS_$STATUS (2) NOT = 0 THEN
      CALL "GDS_$PRINT_STATUS" USING GDS_$STATUS_VECTOR
   END-IF

   COMMIT.

   FINISH.

   STOP RUN.
```

VAX FORTRAN Example

```
   PROGRAM TEST
   include  'dyn_dat1.ftn'
   DATABASE DB = 'example.gdb'
   include  'dyn_dat2.ftn'

   READY
   START_TRANSACTION
```

```
      call GDS_$DDL (gds_$status, DB, gds_$trans,
     +             %VAL (gds_$dyn_length), %REF (gds_$dyn))
      if (gds_$status(2) .ne. 0) call gds_$print_status
(gds_$status)
 COMMIT
 FINISH

 stop
 end
```

The data file, *dyn_dat1.ftn*:

```
      INTEGER*2 GDS_$DYN_LENGTH
      INTEGER*4 GDS_$DYN(12)
```

The data file *dyn_dat2.ftn*:

```
      DATA GDS_$DYN_LENGTH  /48/
      DATA (GDS_$DYN(I)  I=1,12)  /
     +
16910593,5374726,16795974,33556224,1191313410,4719104,
     +   18690,3,117506121,838926418,1543634944,197631/
```

The output from **gdef -d** produces only one filename, which receives the name given on the command line. This file must be split into two files, as shown above. In the FORTRAN program code, the INCLUDE statement for file "one" must be *above* the DB statement; for file "two", below it.

VAX Pascal Example

```
program test (input, output);

DATABASE DB = 'example.gdb'

var
%include 'dyn_dat.pas'

begin

READY;
START_TRANSACTION;

GDS_$DDL (gds_$status, DB, gds_$trans, gds_$dyn_length, %REF
gds_$dyn);
if (gds_$status [2] <> 0) then
```

```
    gds_$print_status (gds_$status);

COMMIT;
FINISH;
end.
```

The data file *dyn_dat.pas*:

```
gds_$dyn_length: gds_$short := 48;
gds_$dyn: packed array [1..48] of char := (
       gds_$dyn_version_1,
           gds_$dyn_begin,
              gds_$dyn_def_rel, chr(1),chr(0), 'R',
                gds_$dyn_end,
              gds_$dyn_def_global_fld, chr(1),chr(0), 'I',
               gds_$dyn_fld_type, chr(2),chr(0), chr(7),chr(0),
             gds_$dyn_fld_length, chr(2),chr(0), chr(2),chr(0),
              gds_$dyn_fld_scale, chr(2),chr(0), chr(0),chr(0),
                gds_$dyn_fld_sub_type, chr(2),chr(0),
chr(0),chr(0),
                gds_$dyn_end,
              gds_$dyn_def_local_fld, chr(1),chr(0), 'I',
                gds_$dyn_rel_name, chr(1),chr(0), 'R',
                gds_$dyn_fld_position, chr(2),chr(0),
chr(0),chr(0),
                gds_$dyn_end,
                gds_$dyn_end,
        gds_$dyn_eoc    );

(* end of DYN string *)
```

VAX PL/I Example

```
DYN_TEST:  PROCEDURE OPTIONS (MAIN);

DATABASE DB = 'example.gdb'

%include 'dyn_dat.pli';

READY;
START_TRANSACTION;

CALL GDS_$DDL (ADDR (gds_$status), DB, gds_$trans,
gds_$dyn_length, gds_$dyn);
```

```
if (gds_$status(2) ~= 0) then
CALL gds_$print_status (gds_$status);

COMMIT;
FINISH;
end;
```

The data file, *dyn_dat.pli*:

```
DECLARE gds_$dyn_length FIXED BINARY (15) STATIC INITIAL (48);
DECLARE gds_$dyn (48) FIXED BINARY (7) STATIC INITIAL (

1,2,9,1,0,82,3,6,1,0,73,70,2,0,7,0,71,2,0,2,0,72,2,0,0,0,73,2,
  0,0,0,3,7,1,0,73,50,1,0,82,92,2,0,0,0,3,3,-1);
```

For More Information

For information on preprocessing, compiling, and linking programs, refer to the chapters on preprocessing your program and compiling and linking your program in the *Programmer's Guide*.

For more information on the **gdef** command, refer to the *DDL Reference*.

Chapter 12
Using Other Interfaces to Define Data

This chapter discusses how to use program interfaces other than **gdef** to define data.

Overview

The interfaces to metadata that InterBase supports are described below. This is followed by a discussion of why you might want to use these interfaces rather than using **gdef**.

Data Definition Alternatives

InterBase supports several interfaces to metadata:

- **Gdef,** which is described in this document.
- **Qli**, the interactive data manipulation utility.

- Host language programs that contain embedded SQL metadata statements. These statements provide a subset of **gdef**'s capabilities. In addition, SQL has its own **grant** and **revoke** statements for securing relations.
- Host language programs that contain dynamic DDL statements. Instructions for generating dynamic DDL statements are presented in Chapter 11, *Modifying Metadata with Dynamic DDL.*
- Host language programs that contain calls to **gds** routines.

You can also write your own interface to metadata by referencing system relations directly. These relations are described in Appendix A, *System Relations.*

Caution

In general, it's best to update metadata by using either **gdef**, **qli** metadata statements, embedded SQL metadata statements, or embedded dynamic DDL statements.

If you choose to update metadata by accessing system relations directly, *be sure you have detailed knowledge of these relations.*

Why Not Use Gdef?

Gdef is a general-purpose metadata definition and modification utility. It accepts either interactive input or input from a file, and can define and modify entities in a database. However, **gdef** may not be appropriate for your application and user population. You may have to read metadata from your application programs, or you might want to allow programs themselves to make certain metadata changes, such as increasing the size of fields.

If you are developing an application for others to use, you may prefer to provide them with a program that updates their database format rather than allowing them to make their own changes with **gdef**.

Because InterBase offers several interfaces to metadata, you can choose the one that best reflects your users' needs.

Metadata Transaction Control

A detailed description of metadata transaction control under **qli** and metadata transaction control in host-language programs is presented below.

Metadata Transaction Control Under Qli

For the sake of efficiency, **qli** maintains its own in-memory image of the database's metadata. When you ask **qli** to store or retrieve data, it uses its in-memory image to verify that your request makes sense. When **qli** readies a database, it creates a new image in-memory of the metadata for the database.

Qli's handling of metadata changes differ, depending on whether you make the changes through **qli** metadata commands or make them directly to the system relations. These differences are discussed below.

Using Qli Metadata Commands

Metadata changes you make by using **qli** metadata update statements like **modify relation** and **modify field** are visible immediately. You can see the changes in the system relations as you make them. The changes become available to other processes when you commit the main transaction.

Metadata changes in **qli** take place under a special-purpose transaction that **qli** starts in response to a metadata update command. **Qli** automatically commits this transaction for you after it performs the requested action.

Changing the System Relations Directly

Metadata changes you make by updating system relations directly are *not* available immediately, because they're not reflected in **qli**'s metadata image.You must not only commit the changes, but also must finish and re-ready the database before you can access the changes. Before **qli** can parse requests involving the new relation, it must re-create its image of the database.

For example, consider the RDB$FIELD_POSITION field of the RDB$RELATION_-FIELDS relation. This field provides **qli** with an ordinal position for displaying field values. The default position for added fields is the highest position plus one, so you may find that newly defined fields are not in the most logical position for display with **qli**. By updating the value of RDB$FIELD_POSITION, you can have **qli** display the fields in the order you want.

The following **qli** script updates the position field:

```
QLI> ready atlas.gdb
QLI> for rdb$relation_fields with
CON>      rdb$relation_name = river_states
CON>      print rdb$field_name, rdb$field_position then
CON>      modify rdb$field_position

            RDB$FIELD              RDB$FIELD
              NAME                 POSITION
=============================== =========
RIVER                                    0
Enter RDB$FIELD_POSITION: 1
STATE                                    1
Enter RDB$FIELD_POSITION: 0

QLI> print first 5 river_states

        RIVER         STATE
==================== =====
Yukon                 AK
Rio Grande            TX
Rio Grande            NM
Rio Grande            CO
Mississippi-Missouri  LA
QLI>
```

As you can see, **qli** has not reversed the field positions, although the following **print** statement shows that the system relations have been updated:

```
QLI> print rdb$field_name, rdb$field_position of
CON>      rdb$relation_fields with
CON>      rdb$relation_name = river_states

            RDB$FIELD              RDB$FIELD
              NAME                 POSITION
=============================== =========

RIVER                                    1
STATE                                    0

QLI>
```

To have **qli** use the new metadata, you must finish the database and re-ready it. The following commands do so and then display the data:

```
QLI> commit
QLI> finish
QLI> ready atlas.gdb
QLI> print first 5 river_states

STATE         RIVER
===== ====================
 AK   Yukon
 TX   Rio Grande
 NM   Rio Grande
 CO   Rio Grande
 LA   Mississippi-Missouri

QLI>
```

Qli waited until the database was detached before providing new metadata. **Qli** reads metadata only once per database *attachment*. This update lag protects the interdependence of many of the system relations, so that you don't have to be concerned with the order of metadata updates.

For more information on updating metadata in **qli**, refer to the chapter on defining metadata in the *Qli Guide*.

Metadata Transaction Control in Host-Language Programs

When you update metadata through a host-language program, the visibility and availability of these changes differs, depending on how you update the metadata:

- When you make metadata changes through SQL statements, you can see the changes in the system relations immediately. These changes become available when you commit the transaction.
- Most metadata changes that you make by updating system relations directly are neither visible (through the system relations) nor available until you finish and re-ready the database.

  Only changes that you make to indexes become available when you commit the associated transaction.

If your metadata changes involve the creation or modification of fields, relations, and views, you need to precompile the program with another database that contains a template of these new objects. If you don't do this, **gpre** won't recognize the new objects, and your program won't precompile.

For more information on updating metadata through SQL statements, refer to the chapter on defining metadata with SQL in the *Programmer's Guide*.

Sample Data Definition Update Program

The following C program adds an index for a relation by adding records to RDB$INDICES and RDB$INDEX_SEGMENTS. The first relation associates the index with a relation, and the latter contains a record for each field that makes up the index.

This example shows that some metadata functions, like adding an index, involve multiple system relations. If you're going to make metadata changes directly, you *must* know about system relation interconnections:

```
/* addindex program */

#include <stdio.h>
#include <ctype.h>

database atlas = "atlas.gdb";

int counter;
char truth [3];
char count [10];

main()
{
  ready;
  start_transaction;

  store ind in rdb$indices using
      printf ("Enter relation name in upper case: ");
      gets (ind.rdb$relation_name);
      printf ("Enter index name in upper case: ");
      gets (ind.rdb$index_name);
      printf ("Is index unique (y/n)? ");
      gets (truth);
      if ( (truth[0] == "y") || (truth[0] == "Y") )
          ind.rdb$unique_flag = 1;
      else
          ind.rdb$unique_flag = 0;
      printf ("How many fields form the index key? ");
      gets (count);
      ind.rdb$segment_count = atoi (count);
      printf ("Enter the key fields from most to least
significant.\n");
      for (counter = 0; counter < ind.rdb$segment_count;
counter++) {
```

```
            store seg in rdb$index_segments using
                strcpy (seg.rdb$index_name, ind.rdb$index_name);
                seg.rdb$field_position = counter;
                printf ("Enter field name in upper case: ");
                gets (seg.rdb$field_name);
            end_store; /* store one segment */
            }
        end_store; /* store the index record */
        commit;
        finish;
    }
```

This example performs an operation in more lines than **gdef** would require. If you choose to write your own metadata interface, you may want to take advantage of host language features, workstation graphics, and alternate input devices (for example, a mouse) to liven up your interface.

In any case, the main thing to remember is that you can use the same language to manipulate both user data and system metadata, so if you find that **gdef** is not an appropriate interface for your application, you can take your knowledge of relational data manipulation and apply it to the metadata.

For More Information

For more information on the metadata interfaces, refer to:

- The *Qli Guide*, for information on using **qli**.
- Chapter 11, *Modifying Metadata with Dynamic DDL*, for information on using host-language programs with dynamic DDL commands.
- The chapter on defining metadata with SQL in the *Programmer's Guide,* for information on using host-language programs with embedded SQL statements.

For more information on system relations, refer to Appendix A, *System Relations*.

Appendix A
System Relations

This appendix lists the InterBase system relations and presents a detailed description of each relation.

Overview

Gdef automatically writes to the InterBase system relations whenever it defines or modifies a data definition. If you use **gdef** for data definition and modification, you don't have to understand these relations.

You can access system relations by using SQL, GDML, **qli**, and the call interface. If your application does require you to read or to write to the system relations, you should pay close attention to the relationships among these relations. For example, a relation in a field appears in two system relations: one that describes global field characteristics, and a second that describes the characteristics of that field as it appears in the relation.

RDB$DATABASE

The RDB$DATABASE system relation defines a database.

Table A-1 describes the RDB$DATABASE relation. All field names in the table are prefixed by the characters RDB$.

Table A-1. RDB$DATABASE

Field Name	Datatype	Length	Description
DESCRIPTION	Blob		Contains a user-written description of the database being defined.
			When you include a comment in a **define** database or **modify database** statement, **gdef** writes to this field.
RELATION_ID	Short		For internal use by InterBase. Do *not* modify.
SECURITY_CLASS	Char	31	Names a security class defined in the RDB$SECURITY_CLASSES relation. The access control limits described in the named security class are applied to all database usage.

RDB$DEPENDENCIES

The RDB$DEPENDENCIES system relation keeps track of the relations and fields that are depended upon by other system objects. These objects can be views, triggers, or computed fields. InterBase uses the RDB$DEPENDENCIES relation to ensure that you can't delete a field or relation that's used in any of these objects.

Table A-2 describes the RDB$DEPENDENCIES relation. All field names in the table are prefixed by the characters RDB$.

Table A-2. RDB$DEPENDENCIES

Field Name	Datatype	Length	Description
OBJECT_NAME	Char	31	Names the object being kept track of in this relation. This object can be a view, trigger, or computed field.
RELATION_NAME	Char	31	Names the relation that's referenced by the object named above.
FIELD_NAME	Char	31	Names the field that's referenced by the object named above.
DEPENDENCY_TYPE	Short		Describes the object type. Valid values are: 0 - view 1 - trigger 2 - computed field All other values are reserved for future use.

RDB$FIELDS

The RDB$FIELDS system relation defines the global characteristics of a field. There is one record in RDB$FIELDS for each global field. Fields are added to relations by means of an entry in the RDB$RELATION_FIELDS relation, where local characteristics are described.

Table A-3 describes the RDB$FIELDS relation. All field names in the table are prefixed by the characters RDB$.

Table A-3. RDB$FIELDS

Field Name	Datatype	Length	Description
FIELD_NAME	Char	31	Names the field defined by this relation. The field name must be unique. If you change the value of this field, you must also change its name in the RDB$FIELD_SOURCE field of any RDB$RELATION_FIELDS relations that include this field.
QUERY_NAME	Char	31	Contains an alternate field name for use in qli; superseded by the query name in RDB$_RELATION_-FIELDS.
VALIDATION_BLR	Blob		For fields with validation criteria, contains the BLR of the validation expression evaluated at time of execution.
VALIDATION	Blob		For fields with validation criteria, contains the original text source expression for the validity check.
COMPUTED_BLR	Blob		For computed fields, contains the BLR of the expression the database evaluates at time of execution.
COMPUTED _SOURCE	Blob		For computed fields, contains the original text source expression for the field.

Table A-3. RDB$FIELDS continued

Field Name	Datatype	Length	Description
DEFAULT_VALUE	Blob		This field is reserved for future use.
FIELD_LENGTH	Short		Contains the length of the field defined in this record. Non-char field lengths are: short - 2 long - 4 quad - 8 float - 4 d_float - 8 double - 8 date - 8 blob - 8
FIELD_SCALE	Short		Contains the scale factor for integer datatypes. The scale factor is the power of 10 by which the integer is multiplied.

Table A-3. RDB$FIELDS continued

Field Name	Datatype	Length	Description
FIELD_TYPE	Short		Specifies the datatype of the field being defined. Changing the value of this system field automatically changes the datatype for all fields based on the field being defined. Valid values are: short - 7 long - 8 quad - 9 float - 10 d_float - 11 char - 14 double - 27 date - 35 varying char - 37 'C' string (null terminated text) - 40 blob - 261 Restrictions: The value of this field can't be changed to or from a blob. Non-numeric data causes a conversion error in a field changed from char to numeric. Changing data from char to numeric and back again adversely affects index performance. It's best to delete and re-create indexes when you make this type of change.

Table A-3. RDB$FIELDS continued

Field Name	Datatype	Length	Description
FIELD_SUB_TYPE	Short		Used by **qli** and **gpre** to distinguish types of blobs and text. Predefined subtypes for blob fields are: 0 - unspecified 1 - char 2 - BLR 3 - access control list 4 - reserved for future use 5 - an encoded description of the current metadata for a relation 6 - a description of a multi-database transaction that finished irregularly Predefined subtypes for char fields are: 0 - unspecified 1 - fixed binary data
MISSING_VALUE	Blob		Contains the BLR for the missing value for this field.
DESCRIPTION	Blob		Contains a user-written description of the field being defined. When you include a comment in a **define field** or modify field statement, **gdef** writes to this field.
QUERY_HEADER	Blob		Contains an alternate column header for use in **qli**; superseded by the query header in RDB$RELATION_FIELDS.
SEGMENT_ LENGTH	Short		Used for blob fields only; a non-binding suggestion for the length of blob buffers.

Table A-3. RDB$FIELDS continued

Field Name	Datatype	Length	Description
EDIT_STRING	Char	125	Contains formatting information for use in qli; superseded by the edit string in RDB$RELATION_-FIELDS.
EXTERNAL_ LENGTH	Short		Indicates the length of the field as it exists in an external relation. If the field is not in an external relation, this value is 0.
EXTERNAL_ SCALE	Short		Indicates the scale factor for an external field that has an integer datatype. The scale factor is the power of 10 by which the integer is multiplied.
EXTERNAL_TYPE	Short		Indicates the datatype of the field as it exists in an external relation. Valid values are: short - 7 long - 8 quad - 9 float - 10 d_float - 11 char - 14 double - 27 date - 35 varying char - 37 'C' string (null terminated text) - 40 blob - 261
DIMENSIONS	Short		For an array datatype, specifies the number of dimensions in the array. For a non-array field, the value is 0.

RDB$FIELD_DIMENSIONS

The RDB$FIELD_DIMENSIONS system relation describes each dimension of an array field.

Table A-4 describes the RDB$FIELD_DIMENSIONS relation. All field names in the table are prefixed by the characters RDB$.

Table A-4. RDB$FIELD_DIMENSIONS

Field Name	Datatype	Length	Description
FIELD_NAME	Short		Names the array field described by this relation. The field name must exist in the RDB$FIELD_NAME field of RDB$FIELDS.
DIMENSION	Short		Identifies one dimension of the array field. The first dimension is identified by the integer 0.
LOWER_BOUND	Long		Indicates the lower bound of the dimension identified above.
UPPER_BOUND	Long		Indicates the upper bound of the dimension identified above.

RDB$FILES

The RDB$FILES system relation lists the secondary files and shadow files for a database.

Table A-5 describes the RDB$FILES relation. All field names in the table are prefixed by the characters RDB$.

Table A-5. RDB$FILES

Field Name	Datatype	Length	Description
FILE_NAME	Char	125	Names either a secondary file or a shadow file for the database.
FILE_SEQUENCE	Short		Specifies either the order that secondary files are to be used in the database or the order of files within a shadow set.
FILE_START	Long		Specifies the starting page number for a secondary file or shadow file.
FILE_LENGTH	Long		Specifies the file length in blocks.
FILE_FLAGS	Short		Reserved for system use.
SHADOW	Short		Specifies the set number of a shadow file. This indicates which shadow set the file belongs to. If the value of this field is 0 or missing, InterBase assumes the file being defined is a secondary file, not a shadow file.

RDB$FILTERS

The RDB$FILTERS relation defines a blob filter.

Table A-6 describes the RDB$FILTERS relation. All field names in the table are prefixed by the characters RDB$.

Table A-6. RDB$FILTERS

Field Name	Datatype	Length	Description
FUNCTION_NAME	Char	31	Names the filter defined by this record. The filter name must be unique.
DESCRIPTION	Blob		Contains a user-written description of the filter being defined. When you include a comment in a **define filter** statement, **gdef** writes to this field.
MODULE_NAME	Char	31	Names the library where the filter executable is stored.
ENTRYPOINT	Char	31	Specifies the entry point within the filter library for the blob filter being defined.
INPUT_SUB_TYPE	Short		Specifies the blob subtype of the input data.
OUTPUT_SUB_TYPE	Short		Specifies the blob subtype of the output data.

RDB$FORMATS

The RDB$FORMATS relation keeps track of the formats of the fields in a relation. InterBase assigns the relation a new format number each time a field definition is changed. This allows existing application programs to access a changed relation, without the need to be recompiled.

Table A-7 describes the RDB$FORMATS relation. All field names in the table are prefixed by the characters RDB$.

Table A-7. RDB$FORMATS

Field Name	Datatype	Length	Description
RELATION_ID	Short		Names a relation that exists in RDB$RELATIONS.
FORMAT	Short		Specifies the format number of the relation. A relation can have any number of different formats, depending on how many times the relation was updated.
DESCRIPTOR	Blob		Lists each field in the relation, along with its datatype and length and scale (if applicable).

RDB$FUNCTIONS

The RDB$FUNCTIONS system relation defines a user-defined function.

Table A-8 describes the RDB$FUNCTIONS relation. All field names in the table are prefixed by the characters RDB$.

Table A-8. RDB$FUNCTIONS

Field Name	Datatype	Length	Description
FUNCTION_NAME	Char	31	Names the function defined by this record. The function name must be unique.
FUNCTION_TYPE	Short		Reserved for future use.
QUERY_NAME	Char	31	Specifies an alternate name for the function that can be used in **qli**.
DESCRIPTION	Blob		Contains a user-written description of the function being defined. When you include a comment in a **define** function statement, **gdef** writes to this field.
MODULE_NAME	Char	31	Names the function library where the function executable is stored.
ENTRYPOINT	Char	31	Specifies the entry point within the function library for the function being defined.
RETURN	Short		Specifies the position of the argument that gets returned to the calling program. This position is specified in relation to other arguments.

RDB$FUNCTION_ARGUMENTS

The RDB$FUNCTION_ARGUMENTS system relation defines the attributes of a function argument.

Table A-9 describes the RDB$FUNCTION_ARGUMENTS relation. All field names in the table are prefixed by the characters RDB$.

Table A-9. RDB$FUNCTION_ARGUMENTS

Field Name	Datatype	Length	Description
FUNCTION_NAME	Char	31	Names the function with which the argument is associated. The function name must be unique and must correspond to a function name in RDB$-FUNCTIONS.
ARGUMENT	Short		Specifies the position of the argument being defined in relation to the other arguments.
MECHANISM	Short		Specifies whether the argument is passed by value (value of 0) or by reference (value of 1).
FIELD_TYPE	Short		Specifies the datatype of the argument being defined. Valid values are: short - 7 long - 8 quad - 9 float - 10 d_float - 11 char - 14 double - 27 date - 35 varying char - 37 'C' string (null terminated text) - 40 blob - 261

Table A-9. RDB$FUNCTION_ARGUMENTS continued

Field Name	Datatype	Length	Description
FIELD_SCALE	Short		Specifies the scale factor for an argument that has an integer datatype. The scale factor is the power of 10 by which the integer is multiplied.
FIELD_LENGTH	Short		Contains the length of the argument defined in this record. Field lengths are: short - 2 long - 4 quad - 8 float - 4 d_float - 8 double - 8 date - 8 blob - 8
FIELD_SUBTYPE	Short		Reserved for future use.

RDB$GENERATORS

The RDB$GENERATORS system relation provides the ability to generate a unique identifier for a relation.

Table A-10 describes the RDB$GENERATORS relation. All field names in the table are prefixed by the characters RDB$.

Table A-10. RDB$GENERATORS

Field Name	Datatype	Length	Description
GENERATOR_NAME	Char	31	Names the relation for which a unique identifier is to be generated.
GENERATOR_ID	Short		Specifies the increment value for the unique identifier.
SYSTEM_FLAG	Short		Indicates whether the relation contains user-data (value of 0) or system information (value greater than 0).

RDB$INDEX_SEGMENTS

The RDB$INDEX_SEGMENTS system relation specifies the fields that comprise an index for a relation. Modifying these records corrupts rather than changes an index unless you delete and re-create the RDB$INDICES record in the same transaction. You can't modify an index, except to make it inactive.

Table A-11 describes the RDB$INDEX_SEGMENTS relation. All field names in the table are prefixed by the characters RDB$.

Table A-11. RDB$INDEX_SEGMENTS

Field Name	Datatype	Length	Description
INDEX_NAME	Char	31	Specifies the index associated with this index segment. Changes to the value of this field must also be made to the RDB$INDEX_NAME field in RDB$INDICES.
FIELD_NAME	Char	31	Names the index segment being defined. The value of this field must match the value of the RDB$FIELD_NAME field in RDB$RELATION_FIELDS.
FIELD_POSITION	Short		Specifies the position of the index segment being defined. The position corresponds to the sort order of the index.

RDB$INDICES

The RDB$INDICES relation defines the index structures that allow InterBase to locate records in the database more quickly. Because InterBase allows you to create both simple indexes (a single key field) and multi-segment indexes (multiple key fields), each index defined in this relation must have corresponding occurrences in the RDB$INDEX_SEGMENTS relation.

Table A-12 describes the RDB$INDICES relation. All field names in the table are prefixed by the characters RDB$.

Table A-12. RDB$INDICES

Field Name	Datatype	Length	Description
INDEX_NAME	Char	31	Names the index being defined. Changes to the value of this field must also be made in the RDB$INDEX_SEGMENTS relation.
RELATION_NAME	Char	31	Names the relation with which this index is associated. The relation must be defined in RDB$RELATIONS.
INDEX_ID	Short		Contains an internal identifier for the index being defined. Do *not* write to this field.
UNIQUE_FLAG	Short		Specifies whether the index allows duplicate values (value of 0) or not (value of 1). Duplicates must be eliminated before a unique index can be created.
DESCRIPTION	Blob		Contains a user-written description of the index being defined. When you include a comment in a **define index** or modify index state-ment, **gdef** writes to this field.
SEGMENT_COUNT	Short		Specifies the number of segments in the index. If the index is a simple index, this field has a value of 1.

Table A-12. RDB$INDICES continued

Field Name	Datatype	Length	Description
INDEX_INACTIVE	Short		Specifies whether the index is active (0) or inactive (2).
INDEX_TYPE	Short		This field is reserved for future use.

RDB$PAGES

The RDB$PAGES system relation keeps track of each page allocated to the databbase. Modifying this relation in any way corrupts your database.

Table A-13 describes the RDB$PAGES relation. All field names in the table are prefixed by the characters RDB$.

Table A-13. RDB$PAGES

Field Name	Datatype	Length	Description
PAGE_NUMBER	Long		Indicates the physically allocated page number.
RELATION_ID	Short		Indicates the id number of the relation for which this page is allocated.
PAGE_SEQUENCE	Long		Indicates the sequence number of this page in relation to other pages allocated for the relation identified above.
PAGE_TYPE	Short		Describes the type of page. This information is for system use only.

RDB$RELATIONS

The RDB$RELATIONS system relation defines some of the characteristics of relations and views. Other characteristics, such as the fields included in the relation and a description of each field, are stored in the RDB$RELATION_FIELDS and RDB$FIELDS relations, respectively.

Table A-14 describes the RDB$RELATIONS relation. All field names in the table are prefixed by the characters RDB$.

Table A-14. RDB$RELATIONS

Field Name	Datatype	Length	Description
RDB$VIEW_BLR	Blob		For a view, contains the BLR of the record selection expression InterBase evaluates at the time of execution.
RDB$VIEW_SOURCE	Blob		For a view, contains the original source record selection expression of the view definition.
RDB$DESCRIPTION	Blob		Contains a user-written description of the relation being defined. When you include a comment in a **define** relation or **modify relation** statement, **gdef** writes to this field.
RDB$RELATION_ID	Short		Contains the internal identification number used in BLR requests. Do *not* modify this field.
RDB$SYSTEM_FLAG	Short		Indicates whether the relation being defined contains user data (value of 0) or system information (value of 1). Do *not* set this field to 1 for relations that you create.

Table A-14. RDB$RELATIONS continued

Field Name	Datatype	Length	Description
DBKEY_LENGTH	Short		Indicates the length of the database key. This field has a value of 8 for relations, and 8 times the number of relations included in the view for views. Do *not* modify the value of this field.
FORMAT	Short		For InterBase internal use only. Do *not* modify.
FIELD_ID	Short		Specifies the number of fields in the relation. This field is maintained by InterBase. Do *not* modify the value of this field.
RELATION_NAME	Char	31	Contains the unique name of the relation defined by this record.
			Changes to the value of this field must also be made in the RDB$RELATIONS_NAME field of any RDB$RELATION_-FIELDS, RDB$VIEW_RELATION, and RDB$INDICES relations that include this relation, and in the BLR of any view or trigger that references the relation.
SECURITY_CLASS	Char	31	Names a security class defined in the RDB$SECURITY_CLASSES relation. Access controls defined in the security class will be applied to all uses of this relation.

Table A-14. RDB$RELATIONS continued

Field Name	Datatype	Length	Description
EXTERNAL_FILE	Char	125	Names the file in which the external relation is stored. An external file can be either an Apollo AEGIS stream file or a VAX RMS file. This field is blank if the relation does not correspond to an external file.
RUNTIME	Blob		Describes the metadata for the relation. This field is used to enhance performance. Do *not* modify this field.
EXTERNAL	Blob		Contains a user-written description of the external file.
OWNER_NAME	Char	31	Identifies the creator of the relation or view. The creator is considered the owner for SQL security (**grant/revoke**) purposes.

RDB$RELATION_FIELDS

The RDB$RELATION_FIELDS system relation lists the fields that comprise a relation and describes those local field characteristics that are specific to the relation.

Table A-15 describes the RDB$RELATION_FIELDS relation. All field names in the table are prefixed by the characters RDB$.

Table A-15. RDB$RELATION_FIELDS

Field Name	Datatype	Length	Description
FIELD_NAME	Char	31	Names the field whose local characteristics are being defined in this relation. The combination of this field with that of RDB$RELATION_-NAME must be unique.
RELATION_NAME	Char	31	Names the relation to which a particular field belongs. There must be a relation of this name in RDB$RELATIONS. The combination of the value of this field with the value of the RDB$FIELD field in this relation must be unique.
FIELD_SOURCE	Char	31	Names the related global field in the RDB$FIELDS relation. Modifying this field changes its global characteristics.
QUERY_NAME	Char	31	Contains an alternate field name for use in qli; supersedes the value in RDB$FIELDS.
BASE_FIELD	Char	31	For a view, names the local field in a relation or view that is the base for the view field being defined.

Table A-15. RDB$RELATION_FIELDS continued

Field Name	Datatype	Length	Description
BASE_FIELD			Fields drawn from an input relation specify their origin in RDB$BASE_-FIELD, which gives the local name of the field in the source relation, and RDB$VIEW_CONTEXT, which specifies the location where the base field is found.
EDIT_STRING	Char	125	Contains formatting information for use in qli; supersedes the value in RDB$FIELDS.
FIELD_POSITION	Short		Specifies the position of the field in relation to other fields. **Qli** uses this field when printing records; **gpre** uses the field order for **select** statements. If two or more fields in the same relation have the same value for this field, those fields appear in a random order.
QUERY_HEADER	Blob		Contains an alternate column header for use in **qli**: supersedes the value in RDB$FIELDS.
UPDATE_FLAG	Short		Not used by InterBase; included for compatability with other DSRI-based systems.
FIELD_ID	Short		An identifier that can be used in BLR to name the field. Because this identifier changes when you back up and restore the database with **gbak**, it's best to use it in transient requests only. Do *not* modify this field.

Table A-15. RDB$RELATION_FIELDS continued

Field Name	Datatype	Length	Description
VIEW_CONTEXT	Short		Identifies the context variable used to qualify view fields. It must have the same value as the context variable used in the view BLR for this context stream.
DESCRIPTION	Blob		Contains a user-written description of the field being defined. When you include a field comment in the context of a **define relation** or **modify** relation statement, **gdef** writes to this field.
DEFAULT_VALUE	Blob		This field is reserved for future use.
SYSTEM_FLAG	Short		Indicates whether the field contains user-data (a value of 0) or system information (a value of 1). Do *not* set value to 1 for fields you create.
SECURITY_CLASS	Char	31	Names a security class defined in the RDB$SECURITY_CLASSES relation. The access restrictions that this security class defines are applied to all uses of this field.
COMPLEX_NAME	Char	31	Reserved for future use.

RDB$SECURITY_CLASSES

The RDB$SECURITY_CLASSES system relation defines access control lists and associates them with databases, relations, views, and fields in relations and views.

Table A-16 describes this relation. All field names in the table are prefixed by the characters RDB$.

Table A-16. RDB$SECURITY_CLASSES

Field Name	Datatype	Length	Description
SECURITY_CLASS	Char	31	Names the security class being defined. If you change the value of this field, you must also change its name in the RDB$SECURITY_CLASS field in RDB$_DATABASE, RDB$RELATIONS, and RDB$RELATION_FIELDS.
ACL	Blob		Contains an access control list that specifies users and the privileges granted to those users.
DESCRIPTION	Blob		Contains a user-written description of the security class being defined. When you include a comment in a **define** security_ class statement, **gdef** writes to this field.

RDB$TRANSACTIONS

The RDB$TRANSACTIONS relation keeps track of all multi-database transactions.

Table A-17 describes the RDB$TRANSACTIONS relation. All field names in the table are prefixed by the characters RDB$.

Table A-17. RDB$TRANSACTIONS

Field Name	Datatype	Length	Description
TRANSACTION_ID	Long		Identifies the multi-database transaction being described.
TRANSACTION	Short		Indicates the state of the transaction. Valid values are: 0 - limbo 1 - committed 2 - rolled back
TIMESTAMP	Date		This field is reserved for future use.
TRANSACTION	Blob		Describes a prepared multi-database transaction. This description is made available if the reconnect fails.

RDB$TRIGGERS

The RDB$TRIGGERS system relation defines a trigger.

Table A-18 describes the RDB$TRIGGERS relation. All field names in the table are prefixed by the characters RDB$.

Table A-18. RDB$TRIGGERS

Field Name	Datatype	Length	Description
TRIGGER_NAME	Char	31	Names the trigger being defined.
RELATION_NAME	Char	31	Names the relation that's associated with the trigger being defined. The relation name must exist in RDB$RELATIONS.
TRIGGER	Short		Specifies a sequence number for the trigger being defined. The sequence number determines when a trigger is executed in relation to other triggers of the same type. Triggers that have the same sequence number execute in a random order. If you don't assign a sequence indicator, the trigger is given an indicator of 0.
TRIGGER_TYPE	Short		Specificies the type of trigger being defined. Valid values are: 1 - pre store 2 - post store 3 - pre modify 4 - post modify 5 - pre erase 6 - post erase

Table A-18. RDB$TRIGGERS continued

Field Name	Datatype	Length	Description
TRIGGER_SOURCE	Blob		Contains the original source of the trigger definition. This field is used when you specify a **show triggers** statement through **qli**.
TRIGGER_BLR	Blob		Contains the BLR representation of the trigger source.
DESCRIPTION	Blob		Contains a user-written description of the trigger being defined. When you include a comment in a **define** trigger or **modify trigger** statement, **gdef** writes to this field.
TRIGGER	Short		Indicates whether the trigger being defined is active (value of 0) or inactive (value of 1).
SYSTEM_FLAG	Short		Indicates whether the trigger being defined is a user-defined trigger (value of 0) or a system-defined trigger (value of 1). Do *not* set this field to 1 for triggers you create.

RDB$TRIGGER_MESSAGES

The RDB$TRIGGER_MESSAGES system relation defines a trigger message and associates the message with a particular trigger.

Table A-19 describes the RDB$TRIGGER_MESSAGES relation. All field names in the table are prefixed by the characters RDB$.

Table A-19. RDB$TRIGGER_MESSAGES

Field Name	Datatype	Length	Description
TRIGGER_NAME	Char	31	Names the trigger associated with this trigger message. The trigger name must exist in RDB$TRIGGERS.
MESSAGE_NUMBER	Short		Specifies the message number of the trigger message being defined. The maximum number of messages is 32,767.
MESSAGE	Varying		Contains the source for the trigger message.

RDB$TYPES

The RDB$TYPES relation defines an enumerated data type. This capability is not available in the current release.

Table A-20 describes the RDB$TYPES relation. All field names in the table are prefixed by the characters RDB$.

Table A-20. RDB$TYPES

Field Name	Datatype	Length	Description
FIELD_NAME	Char	31	Names the field for which the enumerated datatype is being defined.
TYPE	Short		Identifies the internal number that represents the field specified above.
TYPE_NAME	Char	31	Specifies the text that corresponds to the internal number.
DESCRIPTION	Blob		Contains a user-written description of the enumerated datatype being defined.
SYSTEM_FLAG	Short		Indicates whether the relation contains user-data (value of 0) or system information (value greater than 0).

RDB$USER_PRIVILEGES

The RDB$USER_PRIVILEGES relation keeps track of the privileges assigned to a user through an SQL **grant** statement. There is one occurrence of this relation for each user/privilege intersection.

Table A-21 describes the RDB$USER_PRIVILEGES relation. All field names in the table are prefixed by the characters RDB$.

Table A-21. RDB$USER_PRIVILEGES

Field Name	Datatype	Length	Description
USER	Char	31	Names the user who was granted the privilege listed in the RDB$PRIVILEGE field, below.
GRANTOR	Char	31	Names the user who granted the privilege.
PRIVILEGE	Char	6	Identifies the privilege granted to the user listed in the RDB$USER field, above. Valid values are: All Select Delete Insert Update
GRANT_OPTION	Short		Indicates whether the privilege was granted with the **with grant option** (value of 1) or not (value of 0). This option enables a user to grant the same authority to other users.
RELATION	Char	31	Identifies the relation to which the privilege applies.
FIELD_NAME	Char	31	For update privileges, identifies the field to which the privilege applies.

RDB$VIEW_RELATIONS

The RDB$VIEW_RELATIONS system relation describes the general characteristics of a view, including the record selection expression that tells InterBase how records should be selected and combined to form the view. Adding a record to RDB$VIEW_RELATIONS does *not* cause a new relation to apear in a view unless you also change the RDB$RELATIONS record that defines the view.

Because view defintions are heavily interdependent, you should delete and re-create views, rather than modify them. When you do this, be sure to change the related entries in RDB$RELATIONS, RDB$RELATION_FIELDS, and RDB$VIEW_RELATIONS.

Table A-22 describes the RDB$VIEW_RELATIONS relation. All field names in the table are prefixed by the characters RDB$.

Table A-22. RDB$VIEW_RELATIONS

Field Name	Datatype	Length	Description
VIEW_NAME	Char	31	Names a view. The combination of RDB$VIEW_NAME and RDB$-VIEW_CONTEXT must be unique.
RELATION_NAME	Char	31	Names a relation used to construct view.
VIEW_CONTEXT	Short		Identifies the context variable used to qualify view fields. This field must have the same value as the context variable used in the view BLR for this context stream.
CONTEXT_NAME	Char	31	Contains a textual version of the context variable identified in RDB$-VIEW_CONTEXT. This variable must match the value of the RDB$VIEW_SOURCE field for the corresponding relation occurrence in RDB$RELATIONS and be unique in the view.

Appendix B
Sample Database Definition

This appendix shows the definition of the *atlas.gdb* database, which is used in many documentation examples.

```
define database "atlas.gdb";

  {The atlas database is the sample database used throughout the
documentation set. It is based on a North American atlas and
gazeteer. Type "show relations" at the QLI prompt for a listing
of the relations in the database.}
  page_size 1024;


/*Global Field Definitions */

define field ALTITUDE          long;
define field AREA              long;
define field AREA_CODE         char [3];
define field AREA_NAME         varying [20];
define field CAPITAL           varying [25];
define field CENSUS_1950       long;
```

```
define field CENSUS_1960      long;
define field CENSUS_1970      long;
define field CENSUS_1980      long;
define field CENTER_FIELD     long;
define field CITY             varying [25];
define field CODE             varying [4];
define field COMMENTS         blob
                               segment_length 60;
define field ELECTED_APPT     char [1]
  valid if (elect_appt = 'E'
    or elected_appt = 'A'
    or elected_appt missing);
ddefine field F1              blob;
define field F2               blob;
define field F3               blob;
define field FIRST_NAME       varying [10];
define field FLAG             char [1]
  valid if (flag = 'Y'
    or flag = 'N'
    or flag missing);
define field GUIDEBOOK        blob
                               segment_length 60;
define field HOME_STADIUM     varying [30];
define field INCORPORATION    date
define field INIT_TERM        date;
define field LAST_NAME        varying [20];
define field LATITUDE         long;
define field LATITUDE_COMPASS char [1]
  missing_value is "x";
define field LATITUDE_DEGREES varying [3]
  missing_value is -1;
define field LATITUDE_MINUTES char [2]
  missing_value is -1;
define field LEAGUE           char [1];
define field LEFT_FIELD       long;
define field LENGTH           long;
define field LOCATION         blob
                               segment_length 60;
define field LONGITUDE        long;
define field MIDDLE_INITIAL   char [1];
define field NAME varying     [20];
define field NUM_TRAILS       long
define field OFFICE           blob
```

```
                                segment_length 40;
define field OUTFLOW           varying [30];
define field PARTY_AFFILIATION char [1];
define field PHONE             char [10]
  edit_string "(xxx)Bxxx-xxxx";
define field POL_TYPE          char [1];
define field POPULATION        long;
define field POSTAL_CODE       char [10];
define field PROVINCE          varying [4];
define field RIGHT_FIELD       long;
define field RIVER             varying [30];
define field SEATING           long;
define field STATE             varying [4];
define field STATEHOOD         date;
define field STATE_NAME        varying [25];
define field SURFACE           char [1];
define field TEAM_NAME         varying [15];
define field TRAILS_LIGHTED    long
  query_name LIT;
define field TRAILS_SET        long;
define field TYPE              char [1]
  valid if (type = 'N' or
    type = 'A' or
    type = 'B');
define field YEAR              char [4];
define field YEAR_FOUNDED      char [4];
define field ZIP               varying [10];

/* Relation Definitions */

define relation BASEBALL_TEAMS
   TEAM_NAME                        position 0,
   CITY                             position 1,
   STATE                            position 2,
   HOME_STADIUM                     position 3,
   LEAGUE                           position 4,
   LEFT_FIELD                       position 5,
   CENTER_FIELD                     position 6,
   RIGHT_FIELD                      position 7,
   SEATING                          position 8,
   SURFACE                          position 9;

define relation CITIES
   CITY                             position 0,
```

```
    STATE                              position 1,
    POPULATION                         position 2,
    ALTITUDE                           position 3,
    LATITUDE_DEGREES                   position 6
      query_name LATD,
    LATITUDE_MINUTES                   position 7
      query_name LATM,
    LATITUDE_COMPASS                   position 8
      query_name LATC,
    LONGITUDE_DEGREES
      based on LATITUDE_DEGREES        position 9
      query_name LONGD,
    LONGITUDE_MINUTES
      based on LATITUDE_MINUTES        position 10
      query_name LONGM,
    LONGITUDE_COMPASS
      based on LATITUDE_COMPASS        position 11
      query_name LONGC,
    LATITUDE
      computed by (latitude_degrees | ' ' |
      latitude_minutes | latitude_compass)position 4
    LONGITUDE
      computed by (longitude_degrees | ' ' |
      longitude_minutes | longitude_compass)position 5;

define relation CROSS_COUNTRY
    AREA_NAME                          position 0,
    CITY                               position 1,
    STATE                              position 2,
    PHONE                              position 3
       edit_string "(xxx)Bxxx-xxxx",
    NUM_TRAILS                         position 4,
    TRAILS_SET                         position 5,
    TRAILS_LIGHTED                     position 6,
    INSTRUCTION
       based on FLAG                   position 7
       query_header "INST",
    RENTALS
       based on FLAG                   position 8
       query_header "RENT",
    REPAIRS
       based on FLAG                   position 9
       query_header "REP",
    FOOD
```

```
        based on FLAG                position 10,
    LODGE
        based on FLAG                position 11
        query_header "BEDS",
    PACKAGES
        based on FLAG                position 12
        query_header "PKG",
    GUIDED_TOURS
        based on FLAG                position 13
        query_header "TOUR",
    COMMENTS                         position 14;

define relation MAYORS
    CITY                             position 0,
    STATE                            position 1,
    PARTY_AFFILIATION                position 3
  query_name PARTY
  query_header "party",
    INIT_TERM                        position 4,
    ELECT_APPT                       position 5,
    FIRST_NAME                       position 6,
    MIDDLE_INITIAL                   position 7,
    LAST_NAME                        position 8,
    MAYOR_NAME
  computed by (first_name | ' ' |
  last_name)                         position 2;

define relation POLITICAL_SUBDIVISIONS
  CODE                               position 0,
  NAME                               position 1,
  AREA                               position 3,
  INCORPORATION                      position 4,
  CAPITAL                            position 5,
  POL_TYPE;


define relation POPULATIONS
    STATE                            position 0,
    CENSUS_1950                      position 1,
    CENSUS_1960                      position 2,
    CENSUS_1970                      position 3,
    CENSUS_1980                      position 4;
```

```
define relation POPULATION_CENTER
    YEAR                                position 0,
    LATITUDE_DEGREES                    position 3,
    LATITUDE_MINUTES                    position 4,
    LATITUDE_COMPASS                    position 5,
    LONGITUDE_DEGREES
  based on LATITUDE_DEGREES             position 6,
    LONGITUDE_MINUTES
  based on LATITUDE_MINUTES             position 7,
    LONGITUDE_COMPASS
  based on LATITUDE_COMPASS             position 8,
    LOCATION                            position 9,
    LATITUDE
  computed by (latitude_degrees | ' ' |
  latitude_minutes | latitude_compass),
    LONGITUDE
   computed by (longitude_degrees | ' ' |
   longitude_minutes | longitude_compass);

define relation PROVINCES
    PROVINCE                            position 0,
    PROVINCE_NAME
  based on STATE_NAME                   position 1,
    AREA                                position 2,
    CAPITAL
   based on CITY                        position 3;


define relation RIVERS
    RIVER                               position 0,
    SOURCE
   based on PROVINCE                    position 1,
    OUTFLOW                             position 2,
    LENGTH                              position 3;

define relation RIVER_STATES
    STATE                               position 0,
    RIVER                               position 1;

define relation SKI_AREAS
    NAME                                position 0,
    TYPE                                position 1,
    CITY                                position 2,
    STATE                               position 3;
```

```
define relation STATES
    STATE                          position 0,
    STATE_NAME                     position 2,
    AREA                           position 3,
    STATEHOOD                      position 4,
    CAPITAL
  based on CITY                    position 5;

define relation TOURISM
    STATE                          position 0,
    ZIP                            position 1,
    CITY                           position 2,
    OFFICE                         position 3,
    GUIDEBOOK                      position 4;

/*View  Definitions          */

define view CITY_TON of c in cities
with c.city matching '*ton*'
  C.CITY                           postion 0,
  C.STATE                          postion 1,
  C.POPULATION                     position 2;

define view LARGE_NON_CAPITALS of s in states
cross c in cities over state
cross cs in cities with cs.state = c.state and
cs.city = s.capital and cs.population < c.population
  C.CITY                           position 0,
  S.STATE_NAME                     position 1,
  S.CAPITAL                        position 2;


define view LT_AVG_CITIES of c in cities
with c.population < average c1.population of c1 in cities
  C.CITY                           position 0,
  C.STATE                          position 1;

define view MIDDLE_AMERICA of c in cities
with c.longitude_degrees between 79 and 104
and c.latitude_degrees between 33 and 42
  C.CITY                           position 0,
  C.STATE                          position 1,
  C.ALTITUDE                       position 2;
```

```
define view POPULATION_DENSITY of p in populations
cross s in states over state
    P.STATE                                  position 0,
    DENSITY_1950
  computed by (p.census_1950/s.area)         position 1,
    DENSITY_1960
  computed by (p.census_1960/s.area)         position 2,
    DENSITY_1970
  computed by (p.census_1970/s.area)         position 3,
    DENSITY_1980
  computed by (p.census_1980/s.area)         position 4;

define view PROVINCE_VIEW of p in political_subdivisions
with p.pol_type = 'P'
  PROVINCE FROM P.CODE                 position 0,
  PROVINCE_NAME FROM P.NAME            position 1,
  P.AREA                               position 2,
  P.CAPITAL                            position 3;

define view SKI_CITIES of s in states
cross ski in ski_areas with s.state = ski.state
  SKI.NAME                             position 0,
  SKI.CITY                             position 1,
  S.STATE_NAME                         position 2;


define view SKI_STATES of c in cross_country
reduced to c.state
  C.STATE                              position 0;

define view SMALLER_CITIES of c in cities
with c.population < 500000
  C.CITY                               position 0,
  C.STATE                              position 1,
  C.POPULATION                         position 2;

define view SMALL_CAPITAL_CITY of s in states
cross c in cities over state
cross cs in cities with cs.state = c.state and
cs.city = s.capital and cs.population < c.population
reduced to s.state, s.caital
  S.STATE_NAME                         position 0,
  S.CAPITAL                            position 1;
```

```
define view SMALL_CITY_TEAMS of b in baseball_team
cross c in cities with b.city = c.city and
b.state = c.state and b.seating > c.population / 10
   C.CITY                          position 0,
   C.STATE                         position 1,
   B.SEATING                       position 2,
   C.POPULATION                    position 3;

define view STATE_VIEW of p in political_subdivisions
with p.pol_type = 'S'
   STATE FROM P.CODE               position 0,
   STATE-NAME FROM P.NAME          postion 1,
   P.AREA                          position 2,
   STATEHOOD FROM P.INCORPORATION  position 3,
   P.CAPITAL                       position 4;

define view VARIED_XC of c in cross_country
with c.comments containing 'varied'
   C.AREA_NAME                     position 0,
   C.STATE                         position 1,
   C.COMMENTS                      position 2;

define view VILLES of c in cities
with c.city containing 'ville'
   C.CITY                          position 0,
   C.STATE                         position 1,
   C.POPULATION                    position 2;

/* Index  Definitions */

define index BBT1 for BASEBALL_TEAMS unique
   TEAM_NAME,
   CITY;

define index DUPE_CITY for CITIES
   STATE;

define index CITIES_1 for CITIES unique
   CITY,
   STATE;

define index MAYORS_1 for MAYORS unique
   CITY,
   STATE;
```

```
define index RIV1 for RIVERS unique
   RIVER,
   SOURCE;

define index STATE_1 for STATES unique
   STATE;

define index XXX for TOURISM unique
   STATE;

/* Trigger Definitions          */

define trigger cascading_store for CROSS_COUNTRY
  pre store 0:
   begin
     if not any c in cities
     with c.city = new.city and c.state = new.state
       store x in cities
          x.city = new.city;
          x.state = new.state;
       end_store;
   end;
end_trigger;
```

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V