

Using Firebird



About IBPhoenix

IBPhoenix is located—everywhere! It is an umbrella company for a network of independent developers around the world who have been involved with the open sourcing of InterBase® 6.0 since the end of 1999, when the proposal was first made by some software engineers who had worked with the code as employees of Borland.

The IBPhoenix team consists of people who have worked with the InterBase® dynasty of databases for many years, in R & D—in the past, as well as currently as members of the independent Firebird project—and as application consultants and engineers.

IBPhoenix exists with a commitment to make Firebird a presence amongst open source database software. It is a great RDBMS which we consider underrated by the market. Besides providing a commercial support network for Firebird and InterBase® users, we are devoted to assisting the user community with documentation and forum-based support. We maintain a bank technical articles and an on-line knowledge database at our website, <http://www.ibphoenix.com>. We continually seek funding opportunities for the Firebird project with a sincere faith in the synergy that a community of dedicated engineers and involved users can bring to Firebird in open source conditions.

This pair of volumes—**Using Firebird** and the **Firebird Reference Guide**—is a digest of information for software engineers and system administrators from many sources in this wonderful community. We hope it is the first of an ongoing series of publications to educate, enlighten and enlarge the rapidly growing community of Firebird users.

Copyright © December, 2002 IBPhoenix Company. All rights reserved. All IBPhoenix products are trademarks or registered trademarks of the IBPhoenix Company. All Firebird products and trademarks are proprietary to the Firebird Project. All Borland and Visibroker products are trademarks or registered trademarks of Borland Software Corporation. Other brand and product names are trademarks or registered trademarks of their respective holders.



Using Firebird

Table of Contents

1 Quick Start Guide	1
What is in the kit?	1
Default disk locations	1
Installing Firebird	4
Installation drives	4
Installation script or program	4
Windows platforms	4
Posix platforms	4
Testing your installation	5
Pinging the server	5
Checking that the Firebird server is running	6
Windows NT4, 2000 and XP	6
Firebird Server and The Guardian	6
Windows Control Panel applet	7
Windows 9x or ME	8
Posix servers	9
Other things you need	10
A network address for the server	10
Default user name and password	11
How to change the SYSDBA password	11
An Admin tool	12
Connecting to the sample database	12
Server name and path	13
The CONNECT statement	13
Using isql	13
Using a GUI client	14
Creating a database using isql	15
Starting isql	15
The CREATE DATABASE statement	15
Performing a client-only install	16
Windows	16
Linux and some other Posix clients	16
Other things worth knowing	17
Firebird SQL	17
String delimiter symbol	17
Double-quoted identifiers	17
Apostrophes in strings	18
Concatenation of strings	18
Division of an integer by an integer	19
Expressions involving NULL	19
Backup	20
How to corrupt a database	21
1. Modifying metadata tables yourself	21
2. Disabling forced writes on Windows	21
Disabling Forced Writes on a Linux server	22
3. Restoring a backup to a running database	22
4. Allowing users to log in during a restore	22
Where to next?	23
How to get help	23
Using these books	23
The Firebird Project	23
2 About Firebird	25
Firebird's origins	25



Using Firebird

Overview of Features	25
Firebird Server	26
Firebird clients	26
Summary of features	27
Comparing Superserver Architecture with Classic	32
Comparison of characteristics	34
Executable and processes	34
Lock management	35
Resource use	35
Local access method	35
Security on Linux and UNIX platforms	36
Which is better?	36
System Requirements	36
Server Memory (all platforms)	36
Disk space	37
Minimum machine specifications	37
Document Conventions	38
3 About Clients and Servers	39
What is a Firebird client?	39
The Firebird client library	41
The server	41
Application development	42
Embedded Firebird applications	43
Pre-defined queries	43
Dynamic applications	43
Component Support for the API	44
Borland RAD environments	44
The Borland Database Engine (BDE)	44
DBExpress™ and Datasnap™	45
Direct-to-API Components	45
Other RAD environments	46
Microsoft "open connectivity"	46
Java	46
API applications	46
The Firebird core API	46
Core API function categories	47
The Services API	47
The Install API	47
Server-side programming	48
Stored procedures	48
Triggers	48
User-defined functions	49
Multi-database applications	49
4 Operating Basics for Firebird Server	50
Running Firebird on Linux/UNIX	50
Starting the server	50
Stopping the server	51
Other ibmgr commands	51
Syntax	51
ibmgr commands and switches	52
START	52
SHUT	52
SHOW	52
USER USER_NAME	52
PASSWORD PASSWORD	52
HELP	53
QUIT	53
Making the server start automatically	53
Sample start/stop script	53
Running Firebird on Windows	54



Using Firebird

Running as a service - NT, 2000 & XP	55
Firebird Server Manager.	55
Running as an application on any Windows platform	56
Starting Firebird as an application manually	56
Stopping the server.	57
Configuration	57
Brief introduction to the command-line tools	57
isql	58
gbak	58
gsec	58
gfix	58
gstat	59
gds_lock_print and iblockpr.	59
ibmgr	59
5 Configuring the Firebird Server	60
Environment variables	60
Where to set environment variables	60
ISC_USER and ISC_PASSWORD	60
INTERBASE.	61
INTERBASE_TMP.	61
INTERBASE_LOCK and INTERBASE_MSG.	61
TMP	61
The Firebird Configuration File	62
Parameters.	62
connection_timeout.	62
cpu_affinity.	63
database_cache_pages	63
deadlock_timeout.	64
dummy_packet_interval.	64
external_function_directory	64
external_file_directory	64
lock_acquire_spins	64
lock_hash_slots.	65
TCP_NO_NAGLE (for Linux only)	65
server_client_mapping	65
server_priority_class	65
server_working_size_max	66
server_working_size_min	66
tmp_directory	66
Parameters not applicable to Firebird	67
Configuring the database cache	67
Calculating the cache size	68
Changing cache size.	69
Verifying cache size	70
6 Network Configuration	72
Network protocols	72
TCP/IP	72
NetBEUI.	72
Local access	72
Mixed platforms.	72
Connecting to a Microsoft Windows Server	73
Local connection	73
String for Windows local connection	73
Caveats for local connections	74
Windows Networking (NetBEUI)	74
Connection string format for clients on a NetBEUI network.	75
TCP/IP on Windows.	75
Connection string format for clients on a Windows TCP/IP network.	75
TCP/IP local connection	75



Using Firebird

Configuring server names and addresses on Windows	76
Inconsistent strings on Windows	76
Connecting to a Linux or UNIX Server	77
Connection string format for clients to a Linux/UNIX server	77
TCP/IP local loopback connection	77
Configuring server names and addresses	78
Configuring the gds_db service on client and server	78
Configuring the Linux/UNIX inetd daemon	79
Testing the connection	79
Subnet restrictions	79
Pinging the server	80
Firewalls	81
7 Troubleshooting Connections	82
Can you connect to a database at all?	82
Attempt a local connection	82
Can you connect to a database on local loopback?	82
Is the server listening on the Firebird port?	83
Attempt to start the server	83
Connection rejected errors	84
Is the database on a physically local drive?	84
Are the user and password valid?	84
Does the server have permissions on the database file?	84
Does the server have permissions to create files in the Firebird install directory?	84
Can the client find the host?	85
Disabling automatic Internet dialup on Windows systems	87
Reorder network adapter bindings	87
Use Internet Explorer configuration	87
Disable autodial in the Registry	88
Disable RAS autodial	88
Prevent RAS from dialing out.	89
8 Transactions in Firebird	90
Context of a transaction	90
Failed COMMITs	91
Rollback never fails	91
Why does Firebird use transactions?	91
The single-transaction model	92
Multiple transactions	93
Cross-Database Transactions	93
Limbo transactions	94
Recovery	94
Record Versions, Concurrency and Isolation Levels	94
Factors affecting concurrency	95
Record Versions.	95
Access Mode	95
Lock resolution	96
WAIT	96
NO WAIT	96
Transaction Isolation.	96
SNAPSHOT	97
SNAPSHOT	97
TABLE STABILITY	97
READ COMMITTED	98
RECORD_VERSION	98
NO RECORD_VERSION	98
Table Reservation	99
Uses of table reservation	99
Parameters for RESERVING	100
Locking and Lock Conflicts	100



Using Firebird

What to expect when lock conflicts occur	101
What is a deadlock?	101
What is a "livelock"?	102
COMMIT with the RETAIN option	102
Programming with Transactions	103
Transactions and the API	103
The Firebird Header File	103
Setting up a transaction parameter buffer (TPB)	104
Setting up and initializing transaction handles	104
API functions pertaining to transactions	105
Starting a transaction	105
Retrieving a transaction ID	105
Stored Procedures, Triggers and Transactions	105
Embedded transactions	105
Stored Procedures	106
Triggers	106
Transaction "Ageing" and Statistics	106
Transaction statistics	106
Transaction "age" and garbage collection	107
Background garbage collection	107
Moving the OAT forward	107
What the statistics can tell you	108
Transactions are taking too long	108
Transactions have died	108
9 Firebird SQL & Queries	110
About the SQL standard	111
Types of SQL statements	111
Procedural language features	112
Embedded language features	112
Static vs Dynamic SQL	113
Statement terminators	113
Interactive SQL (isql)	114
SQL dialects	114
Statement list	115
Queries	116
The SELECT query	117
The INSERT query	117
Form 1:	117
Form 2:	118
The UPDATE query	118
The DELETE query	118
Queries that call stored procedures	119
Sets of data	119
A table is a set	120
Cardinality and degree	120
Output sets	120
Input sets	121
Output sets as input sets	122
Cursor sets	122
Nested sets	123
Sub-selects	123
Correlated sub-selects	124
Predicates	124
Elements of predicates	124
Operators	124
Values	125
NULL predicates	125
NULL in expressions	125
NULL in calculations	126
NULL and user-defined functions	126



Using Firebird

Setting a value to NULL	127
The EXISTS predicate	127
The IN() predicate	128
Limitations	128
JOIN Operations	129
Ambiguity in JOIN queries	129
Qualification by table identifier	129
Qualification by table aliasing	129
SQL-89 JOIN syntax	130
Example	130
SQL-92 (conditional) joins	130
Example	131
Outer joins	131
FULL OUTER JOIN	131
LEFT OUTER JOIN	131
RIGHT OUTER JOIN	132
Cross joins	132
Natural joins	132
Table aliases	132
Example	132
Queries that count rows	133
SELECT COUNT (*) queries	133
Considerations with COUNT()	134
The GROUP BY clause	135
Syntax	135
Input set	136
Grouping column	136
The HAVING sub-clause	137
The COLLATE sub-clause	138
The ORDER BY clause	138
Considerations	138
Examples	139
The UNION operator	140
Union compatible sets	140
UNION ALL	140
The DISTINCT keyword	141
Considerations	141
Column aliasing	141
Internal functions	142
SQL operators and expressions	143
Using expressions to define columns	144
Server context variables	146
Precedence of operators	147
Concatenation operator	147
Arithmetic operators	148
Comparison operators	148
Logical operators	150
Inclusive vs exclusive OR	151
Precedence of logical operations	151
10 Interactive SQL Utility (isql)	152
Modes for isql Use	152
Invoking isql	153
The terminator character	154
Connecting to a database	154
General isql commands	160
BLOBDUMP	161
EDIT	161
EXIT	162
HELP	162
INPUT	163
OUTPUT	164



Using Firebird

QUIT	164	Extracting metadata	193
SHELL	165	Examples using isql -extract	194
SHOW commands	165	Using isql -a	194
SHOW CHECK	166	What isql -[e]xtract does not extract	194
SHOW DATABASE	166	Exiting an interactive isql session	195
SHOW DOMAIN[S]	167	Command-line switches	195
SHOW EXCEPTION[S]	168	Abbreviated commands	196
SHOW FUNCTION[S]	169		
SHOW GENERATOR[S]	170		
SHOW GRANT	171		
SHOW INDEX (SHOW INDICES)	172		
SHOW PROCEDURE[S]	173		
SHOW ROLE[S]	175		
SHOW SQL DIALECT	175		
SHOW SYSTEM	176		
SHOW TABLE[S]	176		
SHOW TRIGGER[S]	177		
SHOW VERSION	179		
SHOW VIEW[S]	180		
SET commands	180		
SET AUTODDL	180		
SET BLOBDISPLAY	181		
SET COUNT	182		
SET ECHO	183		
SET NAMES	185		
SET PLAN	186		
SET PLANONLY	187		
SET SQL DIALECT	187		
SET STATS	188		
SET TERM	189		
SET TIME	190		
SET WARNINGS	191		
		11 Designing Databases	199
		Defining data structures	199
		Some essential terms	199
		Data Definition Language (DDL)	199
		Metadata	199
		The system tables	199
		Design considerations	200
		Description and analysis	200
		Data model <> database	201
		Design goals	201
		Design framework	202
		Analyzing requirements	203
		Collecting and analyzing data	203
		Identifying entities and attributes	204
		Designing tables	207
		Determining unique attributes	208
		Developing a set of rules	208
		Specifying a data type	209
		Choosing international character sets	209
		Specifying domains	210
		Setting default values and NULL status	210
		Defining integrity constraints	211



Using Firebird

Defining CHECK constraints	211
Establishing relationships between objects	212
Enforcing referential integrity	213
Normalizing the database	214
Eliminating repeating groups	214
Removing partly-dependent columns	216
Removing transitively-dependent columns	217
When to break the rules	218
Choosing indexes	219
Increasing cache size	219
Creating a multi-file, distributed database	220
Planning security	220
Naming objects	220
12 Writing and Running Scripts	222
About Firebird scripts	222
Why use scripts?	223
How to create scripts	223
Executing scripts	224
Basic steps	224
Creating a SQL script	225
What is in a DDL script?	226
SQL statements	226
Comments	226
Block comments	226
In-line comments	226
One-line comments	227
isql statements	227
Terminator symbols	227
Terminators and procedure language (PSQL)	228
Committing Statements in Scripts	230
DDL statements	230
DML statements	230
13 Firebird Data Types	231
About Firebird data types	231
Supported data types	231
BLOBs	232
Arrays	232
Where to specify data types	232
Fixed-decimal (scaled) data types	233
NUMERIC and DECIMAL with precision and scale	235
Numeric input and exponents	236
Integer data types: SMALLINT, INTEGER and INT64	236
Fixed-decimal data types: NUMERIC and DECIMAL	237
NUMERIC data type	237
DECIMAL data type	238
Using exact numeric data types in arithmetic	238
Operations using exact numerics	239
Addition and subtraction	239
Multiplication	240
Division	240
Specifying numeric data types in embedded applications	241
Floating-point data types	242
Example using numerical data types	243
The DATE, TIME, and TIMESTAMP data types	243
Date and time value context variables	244
Date literals	245
Recognized formats for date literals	245
Predefined date literals	249
The TIME data type	249
Firebird's "sliding century window"	250



Using Firebird

Operations using date and time values	251
Converting to the DATE, TIME, and TIMESTAMP data types	253
Casting between time and date types	254
The CAST() function	254
Valid CASTs	255
Casting from other data types to date and time types	255
Casting from date and time datatypes to other SQL datatypes	256
Character data types	257
Specifying a character set	257
Characters vs. bytes	258
Using CHARACTER SET NONE	258
Transliteration errors	259
About collation order	259
Fixed-length character data	259
CHAR(n) or CHARACTER(n)	259
NCHAR(n) or NATIONAL CHAR(n)	260
Variable-length character data	261
VARCHAR(n)	261
NCHAR VARYING(n)	261
Defining BLOB data types	262
BLOB columns	262
BLOB segment length	263
Defining segment length	264
Segment syntax	264
BLOB subtypes	264
Order of clauses	265
BLOB filters	265
Viewing BLOBs	266
Firebird arrays	266
When to use an array type	266
Element data types	266
Defining arrays	266
Multi-dimensional arrays	267
Specifying subscript ranges for array dimensions	267
Custom (explicit) subscript boundaries	268
Converting data types	269
Implicit type conversions	269
Explicit type conversions: CAST()	270
Changing column and domain definitions	270
Altering columns in tables	270
Altering domains	271
Constraints on altering data types	271
14 Databases	272
What you should know	272
Database object naming conventions	273
Delimited identifiers	273
Database file-naming conventions	274
Creating a database	274
Using CREATE DATABASE	275
Syntax	275
Creating a single-file database	276
Specifying file size for a single-file database	277
Multi-file databases	277
Creating a multi-file database	278
Size parameters for multiple files	278
Specifying a secondary file using LENGTH	278
Specifying the starting page number of a secondary file	279
Specifying user name and password	279



Using Firebird

Specifying database page size	280
When to increase page size	280
Changing page size for an existing database	280
Specifying the default character set	281
Using CHARACTER SET NONE	281
Connecting to a database	282
Getting exclusive access to a database	283
About exclusive access	283
'Single-user mode'	283
Restoring multiple-user access	284
More about databases	284
15 Domains and Generators	285
Firebird domains	285
Domain overrides.	286
Syntax for CREATE DOMAIN	286
Using CREATE DOMAIN	286
Domain identifier	286
Data type for the domain	287
The DEFAULT attribute	288
When defaults won't work	288
Overriding DEFAULT	289
Example	289
The NOT NULL attribute	289
CHECK data conditions	290
Syntax for CHECK constraints	290
The VALUE keyword.	290
The CHARSET/CHARACTER SET attribute	292
Column-level override.	292
The COLLATE attribute	292
Column-level override.	292
Altering domains	293
The ALTER DOMAIN statement	293
Syntax	293
Dropping a domain	295
Syntax	295
Firebird Generators	296
Generators vs autoincrement types	296
How generators are used.	296
Creating generators	297
Syntax	297
Generators and data types	297
Using a generator	298
Fetching the previous value	298
Using GEN_ID() to decrement the generator.	299
Retrieving a generator value into an application	299
Setting or resetting generator values	299
Syntax for SET GENERATOR	299
Deleting a generator	300
Syntax for DROP GENERATOR	300
16 Character Sets and Collation Orders	301
About...	301
Character sets.	301
Naming of character sets.	301
Aliases	302
Collation orders	302
Naming of collation orders	302
Default character set	302
Defaults per database element	303
Newly created database	303
Connection	303
Statement	303



Using Firebird

Character set marker	303
String literal	303
Character set storage requirements	304
Collation orders and index key size	304
Special character sets	306
NONE, OCTETS, and ASCII	306
ISO8859_1 (LATIN_1) and WIN1252	306
Support for Paradox and dBASE	307
Character sets for DOS	307
Character sets for Microsoft Windows	308
Additional character sets and collations	308
Transliteration	308
Transliteration errors	308
Specifying character sets	309
Default character set for a database	309
Character set for a column in a table	309
Character set for a client attachment	310
Specifying collation orders	311
Collation order for a column	311
Collation order in comparisons	311
Collation order in ORDER BY	312
Collation order in a GROUP BY clause	312
17 Tables	313
About Firebird tables	313
Physical considerations	313
Structural descriptions	313
System tables	314
Preparing to create tables	314
Table ownership	315
Creating tables	315
Defining columns	315
Required attributes	316
Optional attributes	316
Specifying the datatype	317
Syntax	317
Supported datatypes	318
Defining a character set	318
The COLLATE clause	318
Defining domain-based columns	319
Defining COMPUTED (expression-based) columns	320
Constraints	320
Examples of COMPUTED columns	320
Specifying column default values	321
When defaults won't work	322
Specifying NOT NULL	322
Defining integrity constraints	323
PRIMARY KEY and UNIQUE constraints	323
Atomicity of PRIMARY KEY columns	324
Composite primary keys	324
Declaring the PRIMARY KEY	325
As a column attribute	325
As a named table constraint	325
The UNIQUE constraint	326
UNIQUE keys	326
The FOREIGN KEY constraint	326
Referential integrity	327
Optional referential trigger actions	327
Referring to tables owned by others	329
Orphan rows	330
Circular references	330
Defining a CHECK constraint	331



Using Firebird

Syntax	332
Restrictions	332
Named constraints	333
Syntax	334
Using external files as tables	335
Syntax	335
Uses for the EXTERNAL FILE option	336
Restrictions	336
Importing external files to Firebird tables	337
Casting datatypes	339
Exporting Firebird tables to an external file	340
Altering tables	340
Preparing to use ALTER TABLE	341
Saving existing data	341
An example	342
Dropping columns	342
Using ALTER TABLE	343
Syntax for adding a new column to a table	343
Examples	344
Including integrity constraints	344
Adding new table constraints	344
Dropping an existing column from a table	345
Syntax	345
Dropping existing constraints from a column	345
FOREIGN KEY and PRIMARY KEY constraints	345
CHECK constraints	346
Modifying columns in a table	346
Syntax	346
Examples	347
Constraints on altering data types	347
Summary of ALTER TABLE arguments	349
Removing tables	349
Dropping a table	350
DROP TABLE syntax	350
The RECREATE TABLE statement	350
Syntax	351
Restrictions	351
18 Indexes	352
Index basics	352
When to index	353
Creation of indexes	354
Inspecting indexes	354
Automatic indexing	354
Indexes on PRIMARY KEY columns	354
Indexes on FOREIGN KEY columns	354
Indexes on UNIQUE-constrained columns	355
Duplicating system indexes	355
Using CREATE INDEX	355
Syntax	355
Preventing duplicate entries	355
Existing duplicates	356
Defining a unique index	356
Specifying index sort order	356
When to use a multi-column index	357
OR predicates in queries	357
Examples	357
Improving index performance	359
Housekeeping	359
Using ALTER INDEX	359
Deactivating for batch inserts	360
Syntax	360



Using Firebird

Restrictions	360
Using SET STATISTICS	360
Index selectivity	361
Syntax	361
Restrictions	361
Using DROP INDEX	361
Syntax	362
Restrictions	362
Using gbak for index cleanup	362
19 Views	363
What is a view?	363
Why views can be useful	364
Ways to derive views	365
Creating views	366
View privileges	366
Owner	366
User	366
Syntax	366
Specifying view column names	366
Considerations	367
Using the SELECT statement	367
Using expressions to define columns	368
Read-only and updatable views	368
Naturally updatable views	369
Special consideration for INSERT and UPDATE	369
Changing the behavior of updatable views	369
Examples of read-only views	370
Making read-only views updatable	370
Using WITH CHECK OPTION	372
Examples	373
Modify a view definition?	373
Dropping views	374
Syntax	374
20 More About Databases	375
Read-only databases	375
Making a database read-only	376
Read-only databases with InterBase® 5.x	376
Altering a database	377
Secondary files	377
The ALTER DATABASE statement	377
Syntax	378
Dropping a database	379
Database shadows	379
Benefits of shadowing	380
Limitations of shadowing	380
Before creating a shadow	380
Using CREATE SHADOW	381
Syntax	381
Creating a single-file shadow	382
Shadow location	382
Creating a multi-file shadow	383
Auto mode and manual mode	384
Auto mode	384
Manual mode	385
Conditional shadows	385
Dropping a shadow	385
DROP SHADOW syntax	385
Increasing the size of a shadow	386
Database "hygiene"	386
Background garbage collection	386



Using Firebird

Sweeping	387
Sweep interval	387
Garbage collection during backup	387
The database cache	388
Forced writes	388
Validation and repair	389
Firebird's validation toolset	389
Repairing a corrupt database	389
21 Database Backup and Restore	390
Why back up and restore?	390
gbak backup & restore tool	390
Upgrading the on-disk structure	391
Database backup & restore rights	392
Changing ownership of a database	392
User name and password	393
Running a backup	393
Backing up to a single file	393
Backing up a database to multiple files	393
Arguments for gbak -B[ackup]	394
db_name	394
target	394
size	394
Switches for gbak -B[ackup]	394
-b[ackup_database]	394
-co[nvert]	394
-e[xpand]	394
-fa[ctor] n	394
-g[arbage_collect]	394
-i[ggnore]	395
-l[imbo]	395
-m[etadata]	395
-nt	395
-oI[d_descriptions]	395
-pa[ssword] password	395
-role name	395
-se[rvice] servicename	395
-t[ransportable]	395
-u[ser] name	395
-v[erbose]	395
Return value (Windows only)	396
Target locations for backup files	396
-y [filespec suppress_output]	396
-z	396
Transportable backups	397
Security considerations	397
Running a restore	397
User-defined international objects	398
Multiple-file restores	398
Restoring to a single file	398
Restoring to multiple files	398
Restoring from multiple files	398
Restoring to multiple files from multiple files	398
Arguments for gbak -C[reate] and -R[estore]	399
source	399
db_filename	399
size	399
Switches for gbak -C[reate] and -R[estore]	400
-C[reate_database]	400
-b[uffers]	400
-I[native]	400
-k[iII]	400



Using Firebird

-mo[de] {[read_write read_only]}	400
-n[o_validity]	400
-o[ne_at_a_time]	400
-p[age_size] n	400
pas[sword] password	400
-r[eplace_database]	400
Return value (Windows only)	401
-se[rvice] servicename	401
-u[ser] name	401
-use_[all_space]	401
-v[erbose]	401
-y [filespec suppress_output]	401
-z	401
Database size and performance	402
Using gbak with Firebird Service Manager	402
Backing up with Service Manager	403
Restoring with Service Manager	403
Arguments for the -service switch	403
host_service	403
Restoring on Linux/UNIX	404
Backup and restore examples	404
Backup	404
Restore examples	405
gbak Error messages	406
22 Managing Security	414
Filesystem Security	414
Windows NT/2000 or XP	414
Windows 95/98 and ME	414
False assumptions	415
Linux/UNIX	415
External Objects	416
User-defined functions	416
External files	416
On Windows	417
On Linux/UNIX	417
Password protection	417
isc4.gdb and the masterkey password	417
Vulnerability of isc4.gdb	418
Owner of the server process on Linux/UNIX	419
Web and other n-tier server applications	419
Network Security	419
Sniffers	419
Firewalls	420
Denial-of-Service Attacks	420
Restrictions on use of Firebird tools	421
User restrictions	421
Running gstat on Linux/UNIX	421
Maintaining the security database	421
The gsec utility	421
Starting a gsec interactive session	422
Running gsec remotely	422
Using gsec interactive commands	422
Parameter switches	423
-DATABASE FILESPEC	423
-PW	423
-UID INTEGER	423
-GID INTEGER	423
-FNAME STRING	423
-MNAME STRING	423
-LNAME STRING	423
display	423



Using Firebird

a[dd]	424
mo[dify]	425
de[lete]	425
h[elp] or ?	426
q[uit].	426
Using gsec from a command prompt	426
gsec error messages	427
Database-level security	429
Default security and access	429
Owner	429
SYSDBA and 'Superuser'	429
SQL access privileges	429
Privileges available	430
The ALL keyword	430
The PUBLIC user	431
SQL Roles	431
Implementing roles in a database	431
Granting privileges	432
Granting privileges for a whole table	432
Some examples	434
Granting UPDATE rights for columns	434
Granting privileges for a stored procedure or trigger	434
Multiple privileges and multiple grantees	435
Granting multiple privileges	435
Granting all privileges	436
Granting privileges to multiple users	436
Granting privileges to a list of users	436
Granting privileges to a UNIX group	437
Granting privileges to all users.	437
Granting privileges to a list of procedures	437
Granting privileges through roles	438
Granting privileges to a role	438
Granting a role to users.	439
Granting users the right to grant privileges	440
Restrictions	440
Unintended effects	441
Granting privileges to execute stored procedures	442
Granting privileges for views	443
Using views to restrict data access	443
Updatable vs non-updatable views	444
WITH CHECK OPTION	445
Effects of DML operations on updatable views.	445
Syntax	445
Revoking user access	446
Syntax	447
Restrictions when revoking	448
Revoking multiple privileges	448
Revoking all privileges	448
Users with unknown privileges	449
Revoking privileges for a list of users	449
Revoking privileges for a role	449
To remove privileges from a role	449
To revoke a role from users.	449
Revoking a role from users.	449
Revoking EXECUTE privileges.	450
Revoking privileges from objects	450
Revoking privileges from user PUBLIC	451
Revoking grant authority	451
23 Server and Database Statistics	452
gstat command-line tool	452
gstat on Linux/UNIX.	452
The gstat interface	453



Using Firebird

Syntax	453
Possible switches	453
gstat switches in detail	454
The –data switch	454
Example of data page summary output	454
Restricting the output from gstat –data	455
The –header switch	456
Example of header page summary output	456
The –index switch	460
Example of index page summary output	460
The –r switch	461
Examples of record version output	462
Retrieving statistics into your programs	464
Lock statistics	467
Locking in Firebird	467
The Lock Print utilities	467
Syntax 1	467
Switches for syntax 1	467
Syntax 2	468
Switches for syntax 2	468
24 Housekeeping & Repair: gfix	469
Using gfix	469
Syntax for commands	469
Switches for gfix options	470
Getting database access with gfix	470
User and password on a local connection	471
Shutting down a database	471
Syntax for shutting down	471
Mandatory qualifying switches for shut	471
Terminating a shutdown	472
Syntax	472
Changing database settings	472
Setting default cache size	472
Effect of varying the cache size	473
gfix syntax for setting the cache	473
Other cache options	474
Changing the access mode	474
Syntax for setting access mode	474
Changing the database dialect	474
Syntax for changing the dialect	475
Enabling and disabling 'Use All Space'	475
Enabling and disabling Forced Writes	476
SystemRestore on Windows ME and XP	476
Disabling Forced Writes on Windows servers	477
Working with a converted InterBase 6 database	477
Syntax for enabling and disabling Forced Writes	477
Finding out what version of Firebird server is installed	477
Syntax	477
Transaction recovery	477
Two-phase commit	478
Limbo transactions	478
Transaction recovery	478
Finding limbo transactions	478
Listing limbo transactions and prompting for recovery	479
Requesting automated two-phase recovery	479
Specifically committing or rolling back limbo transactions	479
Data validation and repair	480
When to validate a database	480
Performing a database validation	481
Database repair How-to	481



Using Firebird

Get exclusive access	481
Make a working file copy	482
Perform the validation.	482
Mend corrupted pages	483
Validating after –mend	483
Cleaning and recovering the database	483
Dealing with gbak complications during backup . .	484
Restoring the cleaned backup as a new database . .	484
How to proceed if problems remain	484
Garbage collection and sweeping	485
Garbage collection	485
Sweeping	485
Sweep Interval	486
Considerations for sweeping.	486
Default sweep interval	486
Setting the sweep interval	486
Disabling automatic sweeping	486
Performing a manual sweep	487
Exclusive access for manual sweeps	487
Shadows	488
Activating a shadow	488
Syntax	488
Dropping unavailable shadows	488
Syntax	489
Summary of gfix switches	489
gfix error messages	492
25 Programming on Firebird Server	494
Overview of PSQL code modules	494
Restrictions on PSQL	495
Elements of procedures and triggers	495
Header elements.	496
Body elements	496
Statement types not supported in PSQL	496
Language elements	497
The CREATE statement.	500
BEGIN...END blocks	500
Statement terminator.	501
Using variables	502
The colon (:) marker for variables.	502
Local variables	503
Input parameters.	503
Output parameters	503
NEW and OLD context variables	504
Assignment statements.	505
Using SELECT...INTO statements	506
Singleton SELECTs	506
The WHILE...DO construct	507
The IF...THEN...ELSE construct.	508
Using SUSPEND, EXIT, and END	509
Flow of control effects	510
Example	510
Statements following SUSPEND	511
Error behavior	511
Using generators	512
Event alerters	512
Adding comments.	513
Case-sensitivity and white space	514
Managing compiled objects	514
Managing your code.	514
Editing tools	515



Using Firebird

Compiling stored procedures and triggers	515
Script errors	515
Object dependencies	516
Privileges required for altering objects	516
Steps	516
Testing during development	517
Timing of modifications	517
Choosing your time	517
"Object is in use" error	518
Internals of the technology	519
Benefits of using stored procedures	520
Creating procedures	521
CREATE PROCEDURE syntax	521
Header elements	521
Body elements	522
Selectable stored procedures	526
Multiple-row output	526
SELECTs on stored procedures	526
The FOR SELECT...DO construct	526
Nested and recursive procedures	527
Nested	527
Recursive	527
Altering stored procedures	529
ALTER PROCEDURE syntax	530
The RECREATE PROCEDURE statement	530
Syntax	530
Dropping procedures	530
Syntax	531
Benefits of using triggers	532
Creating triggers	532
CREATE TRIGGER syntax	533
Trigger header elements	534
The trigger body	535
Trigger-specific extension	536
Altering triggers	536
System-defined triggers	536
Syntax	536
Altering a trigger header	537
Altering a trigger body	537
Dropping triggers	538
Syntax	538
Uses for stored procedures	539
Using executable procedures	539
Using selectable procedures	540
Syntax for SELECT from a procedure	540
Examples	541
Using WHERE and ORDER BY clauses	544
Selecting aggregates from procedures	544
Procedures for combined use	545
Viewing arrays with stored procedures	545
Uses for triggers	548
Triggers and transactions	548
Security	549
Error trapping and handling	549
Creating exceptions	550
Altering exceptions	550
Considerations	550
Dropping exceptions	550
Restrictions	551
Raising an exception code	551
Handling exceptions	552



Using Firebird

Syntax for a WHEN...DO statement	553
SQL exceptions	553
Example	553
Firebird-specific errors	554
Exception-handling considerations	555
Example 1	555
Example 2	556
Example 3	557
Example 4—exception handled in a trigger.	559
Handling events on a client	560
How events work	560
The signaler	560
The event manager	561
Responders	561
Event alerters	561
Registering interest in events	562
Registering interest in an event	562
Registering interest in multiple events	563
Asynchronous signaling	563
The AST function	564
Listening for events with EVENT WAIT	564
Responding to events	565
Using RDB\$DB_KEY	566
Size of RDB\$DB_KEY	566
What is RDB\$DB_KEY	567
Usefulness of RDB\$DB_KEY	567
Optimization of queries	568
Benefits	569
Inserting	569
Duration of validity	570
Qualifying RDB\$DB_KEY in the output set	571
RDB\$DB_KEY as a JOIN criterion	571
26 Working with UDFs and Blob Filters	572
About UDFs	572
Existing libraries	572
Developing a UDF	573
Overview	573
Writing a function module	574
Specifying parameters	574
BLOB parameters	575
Specifying a return value	575
Character datatypes	575
Calling conventions	575
Thread-safe UDFs	576
Making UDFs thread-safe	577
Compiling and linking a function module	578
Microsoft Visual C/C++	578
Borland C++	578
Delphi	578
Examples	578
Modifying a UDF library	579
Declaring a UDF to a database	580
Declaration syntax	580
Declaring UDFs with FREE_IT	582
UDF library placement	583
Security of UDF libraries	583
Examples—UDF declaration	583
Finding UDF declaration details	584
Calling a UDF	585
Calling a UDF with SELECT	585
Calling a UDF with INSERT	585



Using Firebird

Calling a UDF with UPDATE	586
Calling a UDF with DELETE	586
Writing a blob UDF	586
Creating a blob control structure	586
Declaring a Blob UDF	588
A blob UDF example	589
The ib_udf libraries	590
ib_udf	590
FBUDF.dll	593
BLOB filters	596
Invoking BLOB filters	597
Pre-defined BLOB filters	597
Using the standard Firebird text filters	597
SUB_TYPE identifiers	597
Custom BLOB filters	598
Declaring a custom BLOB filter	598
Defining a column for a custom BLOB filter	598
Writing an external Blob filter	599
Filter styles	599
Read-only and write-only filters	600
Defining the filter function	600
The blob control structure	602
Setting control structure information field values	604
Programming filter function actions	605
Testing the function return value	608
BLOB descriptors	609
Ways to populate a BLOB descriptor	610
Requesting filtering in an application	611
The BLOB parameter buffer	611
Using API calls to generate a BLOB parameter buffer	611
Generating a BPB directly	613
Expressing numbers in the BPB	615
Making the filter request	615
27 Migrating to Firebird	617
The four Ds	617
The migration checklists	618
Database migration	618
Client migration	618
Migration considerations	618
Dialects	618
Transition features	619
Defence features	619
On-disk structure	620
Servers and databases	621
Clients and databases	621
Reserved words used as identifiers	622
SQL dialects	623
Features available in all dialects	623
Features available only in dialect 3	623
Dialect 1 clients and databases	624
Dialect 2 clients	625
Dialect 3 clients and databases	625
Setting SQL dialect	625
Setting isql client dialect	626
Setting gpre client dialect	627
Setting gfix database dialect	627
New reserved words	628
Reserved word restrictions	628
The double-quotes issue	629



Using Firebird

Single quotes and double quotes: summary	629
DDL and DML statements	629
DATE, TIME, and TIMESTAMP datatypes	630
EXTRACT()	633
Converting TIMESTAMP columns to DATE or TIME	634
Casting date/time datatypes	634
DECIMAL and NUMERIC datatypes	635
Compiled objects	636
Generators	638
Miscellaneous issues	638
Migrating servers and databases	638
“In-place” server migration	639
Migrating to a new server	640
About Firebird 1, dialect 1 databases	641
Transforming databases to Dialect 3	642
Overview	642
Method one: in-place migration	642
Column defaults and column constraints	645
Unnamed table constraints	646
Converting NUMERIC and DECIMAL datatypes	647
Do you really need to migrate your NUMERIC and DECIMAL datatypes?	647
Migrating NUMERIC and DECIMAL datatypes	648
Method two: migrating to a new database	649
Migrating very old databases	650
Migrating clients	651
Migrating data from unrelated DBMS products	652
28 Glossary	653
Aggregate (function) 653	
Alerter (events) 653	
‘ALICE’ 653	
Alternative key (‘alternate key’) 653	
Application Programming Interface (API) 654	
Argument 654	
Array slice 654	
Atomicity 655	
AutoCommit 655	
Backup/restore (Firebird-style) 656	
BDE 656	
Binary tree 657	
BLOB 657	
BLOB Control Structure 657	
BLOB Filter 658	
BLR 658	
Buffer 658	
‘BURP’ 658	
Cache 659	
Cardinality (of an output set) 659	
Case-insensitive index 659	
Cascading integrity constraints 659	
Casting 660	
Character set 660	
Classic architecture 661	
CLOB 661	
Coercing data types 661	
Collation order 661	
Column 662	
Commit 662	
Concurrency 662	
Constraint 663	
Contention 663	
Correlated Subquery 663	
Crash 663	
Crash recovery 663	
Cyclic links 664	



Using Firebird

Database	664	Index	672
DDL	664	Installation	672
Deadlock	664	InterClient	672
Degree (of an output set)	664	InterServer	672
Deployment	664	ISC, isc, etc.	672
DML	664	Isolation level	673
Domain	665	isql	673
DPB	665	JDBC	673
DSQL	665	Join	674
'DUDLEY'	665	Join (inner/outer/left/right)	674
dyn, also DYN	665	jrd	675
Error	666	Key	675
Error Code	666	Kill (shadows)	675
ESQL	666	Leaf bucket	676
Event	666	Limbo (transaction)	676
Exception	666	Locking conflict	676
Execute	667	Lock resolution	677
FIBPlus	667	Metadata	677
Foreign Key	667	Multi-generational architecture (MGA)	677
Garbage collection	668	Natural (scan)	678
gbak	668	Non-standard SQL	678
gdb or GDB	669	Non-unique key	678
GDML	669	Normalization	679
gdef	669	Null	679
Generator	669	ODBC	679
gfix	670	ODS	680
gpre	670	OLAP	680
Grant/Revoke	670	Oldest active transaction	680
gsec	670	Oldest interesting transaction	680
gstat	671	OLE-DB	681
Hierarchical database	671	OLTP	681
Host-language	671	Open source	681
Identity attribute	671	Optimization	682
IBO	671	Outer join/inner join	682
IBX	672	Page	682



Using Firebird

Page_size	682	Snapshot	691
Parameter	683	Snapshot Table Stability	691
Plan	683	Standard SQL	691
Platform	683	Stored Procedure	692
Prepare	684	Subquery	692
Primary Key	684	Sub-select	692
PSQL	684	Superserver architecture	692
qli	685	Surrogate key	693
Query	685	Sweeping	693
RDB\$—	685	System tables	694
RDB\$DB_KEY	685	Table	694
Read Committed	686	Transaction	694
Redundancy	686	Transaction isolation	695
Redundant Indexes	686	Transitively-dependent	695
Referential integrity	687	Trigger	695
Relation	687	Tuple	695
Relationship	687	UDF (“User-defined function”)	696
Replication	688	Unbalanced index	696
Result table	688	Uninstallation	696
Roles	688	Union	697
Rollback	688	Updatable view	697
Schema	688	Validation	697
Schema cache	689	Versioning architecture	698
Scrollable cursor	689	View	698
Selectivity of an index	689	XSQLDA	698
Server Manager	690	XSQLVAR	699
Service Manager	690	Y valve	699
Services API	690		
Sets	690		
Shadowing and shadows	691		

A Alphabetical Index

i

[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

CHAPTER 1

Quick Start Guide

This section is an introduction for the complete newcomer to a few essentials for getting off to a quick start with a Firebird binary kit. For the fine details of configuring and running your server and tuning your installation, please refer to Chapters 4-6 in this volume.

What is in the kit?

All of the kits contain all of the components needed to install the Firebird server:

- the Firebird server executable
- a client library located on the server machine
- the command-line tools
- the standard user-defined function libraries
- a sample database
- the C header files (not needed by beginners!)
- release notes—ESSENTIAL READING!

Default disk locations

The following table describes the default disk locations for the components on Windows and Linux. Information is provided in two version contexts:

- versions prior to Firebird 1.5
- versions from Firebird 1.5 onward

❖ These differences are important. Versions prior to Firebird 1.5 use many of the same locations, component names and resource links as InterBase 6.x and prior versions of InterBase. Hence, it is not possible to run both a Firebird server and an InterBase server on the same machine with these versions.

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

In the major codebase revision starting at v.1.5, the old hard links to InterBase artifacts were removed and many of the major components were renamed. Firebird 1.5 may permit a running InterBase® server to be present. It is a certain objective for Firebird 2.

Table 1: Components of the Firebird installation

Platform	Component	File Name	Default Location
32-bit and 64-bit Windows	Firebird server	Pre v.1.5 ibserver.exe v.1.5 and onward fbserver.exe	c:\Program Files\Firebird\bin
	Command-line tools	gbak.exe, gfix.exe, gstat.exe, etc.	c:\Program Files\Firebird\bin
	Sample database	Employee.gdb	c:\Program Files\Firebird\examples
	User-defined function (UDF) libraries	ib_udf.dll & ib_utils.dll fbudf.dll	c:\Program Files\Firebird\udf
Windows NT, 2000, XP	Firebird client	Pre v.1.5 gds32.dll v.1.5 and onward gds32.dll, linking through to xxxx.dll	c:\Winnt\System32
Windows 9x, ME	Firebird client	As above	c:\Windows



Symbols **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

Table 1: Components of the Firebird installation

Platform	Component	File Name	Default Location
Linux and possibly other UNIX distros	Firebird server	Pre v.1.5 ibserver v.1.5 and onward fbserver	/opt/interbase/bin TBA
	Command-line tools	gbak, gfix, gstat, etc. Pre v.1.5 v.1.5 and onward	/opt/interbase/bin TBA
	Sample database	employee.gdb Pre v.1.5 v.1.5 and onward	/opt/interbase/bin TBA
	UDF libraries	Pre v.1.5 ib_udf.so v.1.5 and onward ib_udf.so fbudf.so	/opt/interbase/udf TBA
	Firebird client	Pre v.1.5 libgds.so.0 (binary) libgds.so (symlink) v.1.5 and onward TBA	/usr/lib



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Installing Firebird

Installation drives

Firebird server—and any databases you create or connect to—must reside on a hard drive that is physically connected to the host machine. You cannot locate components of the server, or any database, on a mapped drive, a filesystem share or a network filesystem.

- ★ You can mount a read-only database on a CD-ROM drive but you cannot run Firebird server from one.

Installation script or program

Although it is possible to install Firebird by a filesystem copying method—such as “untarring” a snapshot build file or decompressing a structured WinZip “.zip” file—it is strongly recommended that you use the distributed release kit the first time you install Firebird. The Windows executable installation script, the Linux rpm (RedHat Package Manager) program and the official tar.gz for other Posix platforms perform some essential setup tasks. Provided you follow the instructions correctly, there should be nothing for you to do upon completion but log in and go!

Windows platforms

On server platforms—Windows NT, 2000 and XP—the Firebird service will be running when the installation completes. Next time you boot up your server machine, it will be started automatically. To find out how to start and stop the server manually, see [Running Firebird on Windows](#) on p. 54 of chapter 4.

The non-server Windows platforms—Windows 95, 98 and ME—do not support services. The installation will start Firebird server as an application, protected by another application known as the Guardian. If the server application should terminate abnormally for some reason, the Guardian will attempt to restart it.

- ★ A copy of the client library—gds32.dll—will have been installed in your system directory.

Posix platforms

In all cases, read the release notes that pertain to the version of Firebird that you are going to install. There may be significant variations from release to release of any Posix operating system, especially the open source ones. Where possible, the build engineers for each Firebird version have attempted to document any known issues.



Symbols [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

If you do not find a copy of the release notes in your kit, go back to the Downloads page of the Firebird website at <http://firebirdsql.org> and download a copy from there.

If you have a Linux distribution that supports **rpm** installs, consult the appropriate platform documentation for instructions about using RedHat Package Manager. In most distributions you will have the choice of performing the install from a command shell or through a GUI interface.

For Linux distributions that cannot process **rpm** programs, and for the various UNIX flavors, use the ".tar.gz" kit. You will find detailed instructions in the release notes.

Shell scripts have been provided. In some cases, the release notes may instruct you to modify the scripts and make some manual adjustments.

Testing your installation

If everything works as designed, the Firebird server process will be running on your server upon completion of the installation. It will start up automatically whenever you restart your server.

At this point, it is assumed that you will use the recommended TCP/IP protocol for your Firebird client/server network.

For information about using NetBEUI protocol in an all-Windows environment, refer to Chapter 6, Network Configuration.

❖ IPX/SX networks are not supported by Firebird

Pinging the server

Usually, the first thing you will want to do once installation is complete is ping the server. This just gives you a reality check to ensure that your client machine is able to see the host machine in your network. For example, if your server's IP address in the domain that is visible to your client is 192.13.14.1, go to a command shell and type the command

`ping 192.13.14.1`

substituting this example IP address for the IP address that your server is broadcasting.



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

- ❖ If you get a timeout message, study Chapter 6, Network Configuration, and Chapter 7, Troubleshooting Connections, for further instructions.

Note that if you are connecting to the server from a local client—that is, a client running on the same machine as the server—you can ping the virtual TCP/IP loopback server:

```
ping localhost -or- ping 127.0.0.1
```

Checking that the Firebird server is running

After installation, Firebird server should be running as a service on Windows NT, 2000 or XP or on Linux.

Windows NT4, 2000 and XP

For the Windows server platforms, start the Services applet on the Control Panel.

Firebird Server and The Guardian

This illustration shows the Services applet display on Windows NT 4. The appearance may vary from one Windows server edition to another.

The Guardian

If the Guardian is running (as shown in the screenshot, over) it may have a different service name because of version changes.

- ★ On Windows 2000 and XP, the Guardian is a convenience rather than a necessity, since these two operating systems have the facility to watch and restart services. It is recommended that you keep Guardian active for other platforms if a SYSDBA is not available to restart the server manually in the event that it is stopped for some reason.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Services

Service	Status	Startup
CesarFTP FTP Server	Started	Automatic
ClipBook Server		Manual
COM+ Event System		Manual
Computer Browser	Started	Automatic
DHCP Client		Disabled
Directory Replicator		Manual
EventLog	Started	Automatic
Firebird Guardian Service	Started	Automatic
Firebird Server	Started	Automatic
Messenger	Started	Automatic

Close

Start

Stop

Pause

Continue

Startup...

Hw Profiles...

Help

Startup Parameters:

Windows Control Panel applet



Earlier distributions of Firebird for Windows kits installed the InterBase Manager applet into the Control Panel of operating systems that support services. It was omitted from Firebird 1.0 because a number of Firebird-specific 'Firebird [Server] Manager' implementations became available through open source channels. Whilst the applet is not essential, it does provide a convenient way to start and stop the service.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z



If you want a Firebird Manager applet and you do not find one installed in the Control Panel of your Windows server, download one from the FirebirdSQL website or from www.ibphoenix.com.

The Control Panel applet will be similar in appearance to the illustration.

Windows 9x or ME

On Windows 9x or ME Firebird server should be running as an application, monitored by the Guardian. The Guardian's icon should appear in the tray with a green graphic. If the icon is flashing or showing as a red graphic, it indicates that Guardian is either attempting to start the server or has failed.

If you used an installation kit that installed but did not automatically start the Guardian and the Firebird server, you can set it up as follows:

- 1 Locate the executable file for the Guardian program (ibguard.exe) and create a shortcut for it in the Startup area of your machine's Start Menu.
- 2 Open the Properties dialog of the shortcut and go to the field where the command line is.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- 3 Edit the command line so it reads as follows:

```
ibguard.exe -a
```

- 4 Save and close the Properties dialog.

- 5 Double-click on the shortcut to start the Guardian. The Guardian will proceed to start ibserver.exe.

The Guardian should start up automatically next time you boot your Windows 9x or ME machine.

❖ There is currently no Control Panel applet which will detect the Firebird server application running on these non-server Windows versions.

Posix servers

Use the **top** command in a command shell to inspect the running processes interactively. If Firebird is running, you should see one process named **ibguard** (this is Guardian), one main and zero or more child processes named **ibserver** or **fbserver**—which one depends on the distribution of Firebird which you have installed.

The following screenshot shows the output of top, restricted by grep to show only processes with names starting with the characters 'ib', i.e.

```
top -A | grep ib
```



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

```
[root@coolduck temp]# top |grep ib
1m 1:25am  up 1 day,  3:42,  1 user,  load average: 0.00, 0.00, 0.00
763 root      9  0   804   804   652 S    0.0  0.1  0:00 ibguard
764 root      9  0  2096  2096  1304 S    0.0  0.4  0:00 ibserver
765 root      9  0  2096  2096  1304 S    0.0  0.4  0:00 ibserver
768 root      9  0  2096  2096  1304 S    0.0  0.4  0:00 ibserver
763 root      9  0   804   804   652 S    0.0  0.1  0:00 ibguard
764 root      9  0  2096  2096  1304 S    0.0  0.4  0:00 ibserver
765 root      9  0  2096  2096  1304 S    0.0  0.4  0:00 ibserver
768 root      9  0  2096  2096  1304 S    0.0  0.4  0:00 ibserver
763 root      9  0   804   804   652 S    0.0  0.1  0:00 ibguard
764 root      9  0  2096  2096  1304 S    0.0  0.4  0:00 ibserver
765 root      9  0  2096  2096  1304 S    0.0  0.4  0:00 ibserver
768 root      9  0  2096  2096  1304 S    0.0  0.4  0:00 ibserver
763 root      9  0   804   804   652 S    0.0  0.1  0:00 ibguard
764 root      9  0  2096  2096  1304 S    0.0  0.4  0:00 ibserver
765 root      9  0  2096  2096  1304 S    0.0  0.4  0:00 ibserver
768 root      9  0  2096  2096  1304 S    0.0  0.4  0:00 ibserver
```

★ If your Firebird server uses the Classic architecture, the process name is `gds_inet_server`. There will be one instance of this process running for each connection. Use `top -A | grep gds` to look at them.

Other things you need

A network address for the server

- If you are on a managed network, get the server's IP address from your system administrator
- If you have a simple network of two machines linked by a crossover cable, you can set up your server with any IP address you like except 127.0.0.1 (which is reserved for a local loopback server) and, of course, the IP address which you are using for your client machine. If you know the "native" IP addresses of your network cards, and they are different, you can simply use those

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

- If you are intending to try out a single-machine installation of both client and server, you should use the local loopback server address—localhost, with the IP address 127.0.0.1

NOTE On Windows, it is possible to connect locally to the server, without using a local TCP/IP loopback. This is not a TCP/IP connection and it is not a thread-safe way to connect to a local server. For using single instances of the command-line tools (`gsec`, `gbak`, etc.) it works just fine.

Default user name and password

The SYSDBA user has all privileges on the server. The installation program will install the SYSDBA user with the password ***masterkey***.

★ Actually, it is *masterke*, since all characters after the eighth one are ignored.

If your server is exposed to the Internet *at all*, you should change this password immediately using the `gsec` command-line utility.

How to change the SYSDBA password

You need to be logged into the operating system as Superuser (root on Linux) to run `gsec`. Let's say you decide to change the SYSDBA password to ***icurry4me***

- 1 Go to a command shell on your server and change to the directory where the command-line utilities are located. Refer to Table 1 (above) to find this location.
- 2 Type the following (it is case-sensitive on all platforms except Windows):

```
gsec -user sysdba -password masterkey
```

You should now see the shell prompt for the `gsec` utility:

GSEC>

- 3 Type this command



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

GSEC> modify sysdba -pw icuryy4me

4 Press ENTER. The new password `icuryy4me` is now encrypted and saved and *masterkey* is no longer valid.

5 Now quit the `gsec` shell:

GSEC> quit

★ Because Firebird ignores all characters in a password past the eighth character, `icuryy4m` will work, as will `icuryy4monkeys`.

An Admin tool

The Firebird kit does not come with a GUI admin tool. It does have a set of command-line tools, executable programs which are located in the `./bin` directory of your Firebird installation.

The range of excellent GUI tools available for use with a Windows client machine is too numerous to describe here. A few GUI tools written in Borland Kylix, for use on Linux client machines, are also in various stages of completion.

Inspect the Downloads > Contributed > Admin Tools page at <http://www.ibphoenix.com> for all of the options.

★ You can use a Windows client to access a Linux server and vice-versa.

Connecting to the sample database

In the `./examples` directory of your Firebird installation is a sample database named `employee.gdb`. You can use this database to "try your wings".

❖ If you are running the server on Windows XP or ME, be sure to rename this test database to `employee.fdb` to get around the System Restore utility, which targets files with a `.gdb` extension. There is more information about this problem in the release notes.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Server name and path

If you move the sample database, be sure you move it to a hard disk that is physically attached to your server machine. Shares, mapped drives or (on Unix) mounted SMB (Samba) filesystems will not work. The same rule applies to any databases that you create.

There are two elements to a TCP/IP connection string: the server name and the disk/filesystem path. Its format is as follows:

For a Linux server: `servername:/filesystem-path/database-file`

Example on a Linux or other Posix server named `serverxyz`:

`serverxyz:/opt/interbase/examples/employee.gdb`

For a Windows server: `servername:D:\filesystem-path\database-file`

`serverxyz:C:\Program Files\Firebird\examples\employee.gdb`

The CONNECT statement

Connecting to a Firebird database always requires the user to “log in” using a user name and a valid password—widely referred to as a *log-in* or *login*. Any user other than SYSDBA or root (on Posix systems only) needs also to have permissions to objects inside a database. For simplicity here, we will look at logging in as SYSDBA using the password *masterkey*.

Using isql

There are several different ways to connect to a database using isql. One way is to start **isql** in its interactive shell:

Go to the `./bin` directory of your Firebird installation and, at that prompt, type the command **isql**

```
C:\Program Files\Firebird\bin>isql [Enter]
```

```
Use CONNECT or CREATE DATABASE to specify a database
```

```
SQL>CONNECT "C:\Program Files\Firebird\examples\Employee.gdb" user  
'SYSDBA' password 'masterkey'; [Enter]
```

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

- ❖ In isql, make sure that every statement ends with a semicolon. If you forget, the prompt will change from SQL> to CON> and you won't be able to continue until you type in the semicolon and press **Enter** again.
- ❖ Although single quote symbols are the "norm" for delimiting strings in Firebird, double quote symbols were used with the database path string in the above example. This is sometimes necessary with some of the command-line utilities where the path string contains spaces. Single quotes should work for paths that do not contain spaces.

At this point, isql will inform you that you are connected:

```
DATABASE "C:\Program Files\Firebird\examples\Employee.gdb", User: sysdba  
SQL>
```

You can now continue to play about with the employee.gdb database. The characters **isql** stand for *interactive SQL [utility]*. You can use it for querying data, getting information about the metadata, creating database objects, running data definition scripts and much more.

To get back to the command prompt type

```
SQL> QUIT;
```

For more information about isql, see chapter 10, [Interactive SQL Utility \(isql\)](#) (p. 152).

Using a GUI client

GUI client tools usually take charge of composing the CONNECT string for you, using server, path, user name and password information that you type into prompting fields. Use the elements as described in the preceding topic.



- It is quite common for such tools to expect the entire server + path as a single string
- Remember that file names and commands on Linux and other Posix command shells are case-sensitive



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Creating a database using isql

There is more than one way to create a database using **isql**. Here, we will look at one simple way to create a database interactively—although, for your serious database definition work, you should create and maintain your metadata objects using data definition scripts. There is a section later in this manual discussing this topic.

Starting isql

To create a database interactively using the **isql** command shell, you need to be working on the server. Get to a command prompt in the **./bin** directory and start **isql** as follows:

```
C:\Program Files\Firebird\bin>isql [Enter]  
Use CONNECT or CREATE DATABASE to specify a database
```

The CREATE DATABASE statement

Now, you can create your new database interactively. Let's suppose that you want to create a database named **test.fdb** and store it a directory named "data" on your D drive:

```
SQL> CREATE DATABASE 'D:\data\test.fdb' page_size 8192 user 'SYSDBA'  
password 'masterkey'; [Enter]
```

The database will be created and, after a few moments, the SQL prompt will reappear. You are now connected to the new database and can proceed to create some test objects in it.

To verify that there really is a database there, type in this query:

```
SQL> SELECT * FROM RDB$RELATIONS; [Enter]
```

The screen will fill up with a large amount of data! This query selects all of the rows in the system table where Firebird stores the metadata for tables. An "empty" database is not empty—it contains a database which will become populated with metadata as you begin creating objects in your database.

To get back to the command prompt type



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

SQL> QUIT;

For more information about isql, see chapter 10, [Interactive SQL Utility \(isql\)](#) (p. 152).

Performing a client-only install

Each remote client machine needs to have the client library—libgds.so on Posix clients, gds32.dll on Windows clients—that matches the release version of the Firebird server.

Firebird versions from 1.5 onward require an additional client library named libfb.so or fb32.dll which contains the full library. In these newer distributions, the “gds” named files are distributed to maintain compatibility with third-party products which need these files. Internally, the libraries will “jump” to the correct access points in the renamed libraries.

Some extra pieces are also needed for the client-only install.

Windows

At the time of publication, no compact installation program was available to assist with installing the client pieces on a Windows client. If you are in the common situation of running Windows clients to a Linux or other Posix Firebird server, you need to download the full Windows installation kit that corresponds to the version of Firebird server you install on your Linux or other server machine.

Fortunately, once you have the kit, the Windows client-only install is easy to do. Run the installation program, just as though you were going to install the server—but select the CLIENT ONLY option from the install menu.

Linux and some other Posix clients

A small-footprint client install program for Linux clients is not available either. Additionally, some Posix flavors—even within the Linux constellation—have somewhat idiosyncratic requirements for filesystem locations. For these reasons, not all *x distributions for Firebird even contain a client-only install option.

For most Linux flavors, the following procedure is suggested for Firebird versions lower than 1.5. Log in as root for this.

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

- 1 Look for libgds.so.0 in /opt/interbase/lib on the server where Firebird server is installed. Copy it to /usr/lib on the client.
- 2 Create the symlink libgds.so for it, using the following command:

```
ln -s /usr/lib/libgds.so.0 /usr/lib/libgds.so
```

- 3 Copy the interbase.msg file to /opt/interbase
- 4 In system-wide default shell profile, or using setenv() from a shell, create the INTERBASE environment variable and point it to /opt/interbase, to enable the API routines to locate the messages.

Other things worth knowing

This chapter finishes with a small, eclectic collection of things other Firebird ‘newbies’ wished they had known about when they began.

Firebird SQL

Every database management system has its own idiosyncrasies in the ways it implements SQL. Firebird adheres to the SQL standard more rigorously than any other RDBMS except possibly its “cousin”, InterBase®. Developers migrating from products that are less standards-compliant often wrongly suppose that Firebird is quirky, whereas many of its apparent quirks are not quirky at all.

String delimiter symbol

Strings in Firebird are delimited by a pair of single quote symbols—'I am a string'—(ASCII code 39 NOT 96). If you used earlier versions of Firebird’s relative, InterBase®, you might recall that double and single quotes were interchangeable as string delimiters. Double quotes cannot be used as string delimiters in Firebird.

Double-quoted identifiers

Before the SQL-92 standard, it was not legal to have object names (identifiers) in a database that duplicated keywords in the language, were case-sensitive or contained spaces. SQL-92 introduced a single new standard



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

to make any of them legal, provided that the identifiers were defined within pairs of double-quote symbols (ASCII 34) and were always referred to using double-quote delimiters.

The purpose of this “gift” was to make it easier to migrate metadata from non-standard RDBMSs to standards-compliant ones. The down-side is that, if you choose to define an identifier in double quotes, its case-sensitivity and the enforced double-quoting will remain mandatory.

Firebird does permit a slight relaxation under a very limited set of conditions: if the identifier which was defined in double-quotes 1) was defined as all upper-case, 2) is not a keyword and 3) does not contain any spaces, then it can be used in SQL unquoted as long as it is used in all upper-case.

Unless you have a compelling reason to define quoted identifiers, it is usually recommended that you avoid them. Firebird happily accepts a mix of quoted and unquoted identifiers—so there is no problem including that keyword which you inherited from a legacy database, if you need to.

- ❖ Some database admin tools enforce double-quoting of ALL identifiers by default. Try to choose a tool which makes double-quoting optional.

Apostrophes in strings

If you need to use an apostrophe inside a Firebird string, you can “escape” the apostrophe character by preceding it with another apostrophe.

For example, this string will give an error:

```
'Joe 's Emporium'
```

because the parser encounters the apostrophe and interprets the string as ‘Joe’ followed by some unknown keywords.

To make this a legal string, double the apostrophe character:

```
'Joe ''s Emporium'
```

Notice that this is TWO single quotes, not one double-quote.

Concatenation of strings

The concatenation symbol in SQL is two ‘pipe’ symbols (ASCII 124, in a pair with no space between). In SQL, the ‘+’ symbol is an arithmetic operator and it will cause an error if you attempt to use it for concatenating strings. The following expression prefixes a character column value with the characters ‘Reported by::’:

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

'Reported by: ' || LastName

Take care with concatenations. Be aware that Firebird will raise an error if your expression attempts to concatenate two or more char or varchar columns whose potential combined lengths would exceed the maximum length limit for a char or a varchar (32 Kb).

See also the note below, [Expressions involving NULL](#), about concatenating in expressions involving NULL.

Division of an integer by an integer

Firebird accords with the SQL standard by truncating the result (quotient) of an integer/integer calculation to the next lower integer. This can have bizarre results unless you are aware of it.

For example, this calculation is correct in SQL:

1 / 3 = 0

If you are upgrading from a RDBMS which resolves integer/integer division to a float quotient, you will need to alter any affected expressions to use a float or scaled numeric type for either dividend, divisor, or both.

For example, the calculation above could be modified thus in order to produce a non-zero result:

1.000 / 3 = 0.333

Expressions involving NULL

In SQL, NULL is not a value. It is a condition, or *state*, of a data item, in which its value is *unknown*. Because it is unknown, null cannot behave like a value. When you try to perform arithmetic on NULL, or involve it with values in other expressions, the result of the operation will always be NULL. It is not zero or blank or an "empty string" and it does not behave like any of these values.

So—here are some examples of the types of surprises you will get if you try to perform calculations and comparisons with NULL:

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

```
1 + 2 + 3 + NULL = NULL  
if (a = b) then  
MyVariable = 'Equal'  
else  
MyVariable = 'Not equal';  
will return 'Not equal' if both a and b are null.
```

```
if (a <> b) then  
MyVariable = 'Not equal'  
else  
MyVariable = 'Equal';  
will also return 'Not equal' if both a and b are null.
```

```
FirstName || ' ' || LastName  
will return NULL if either FirstName or LastName is NULL.
```

Backup

Firebird comes with its own utility for backing up and restoring your databases. Its name is **gbak** and it can be found in the ./bin directory of your Firebird installation. Firebird databases can be backed up whilst users are connected to the system and going about their normal work. The backup will be taken from a snapshot of the database state at the time the backup began.

Regular backups and occasional restores using **gbak** should be a scheduled part of your database management activity.

- ❖ Do not use external proprietary backup utilities or file-copying tools such as WinZip, tar, copy, xcopy, etc., on a database which is running. Not only will the backup be unreliable, but the disk-level blocking used by these tools can corrupt a running database.
- ⚠ Study the warnings in the next section about database activity during restores!



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

How to corrupt a database

1. Modifying metadata tables yourself

Firebird stores and maintains all of the metadata for its own and your user-defined objects in—a Firebird database! More precisely, it stores them in relations (tables) right in the database itself. The identifiers for the system tables, their columns and several other types of system objects begin with the characters 'RDB\$'.

Because these are ordinary database objects, they can be queried and manipulated just like your user-defined objects. However, just because you *can* does not say you *should*. The Firebird engine implements a high-level subset of SQL (DDL) for the purpose of defining and operating on metadata objects, typically through CREATE, ALTER and DROP statements.

It cannot be recommended too strongly that you use DDL—not direct SQL operations on the system tables—whenever you need to alter or remove metadata. Defer the “hot fix” stuff until your skills in SQL and your knowledge of the Firebird engine become very advanced. A wrecked database is neither pretty to behold nor cheap to repair.

2. Disabling forced writes on Windows

Firebird is installed with forced writes (synchronous writes) enabled by default. Changed and new data are written to disk immediately upon posting.

It is possible to configure a database to use asynchronous data writes—whereby modified or new data are held in the memory cache for periodic flushing to disk by the operating system’s I/O subsystem. The common term for this configuration is *forced writes off* (or *disabled*). It is sometimes resorted to in order to improve performance during large batch operations.

The big warning here is—**do not disable forced writes on a Windows server**. It has been observed that the Windows server platforms do not flush the write cache until the Firebird service is shut down. Apart from power interruptions, there is just too much that can go wrong on a Windows server. If it should hang, the I/O system goes out of reach and your users’ work will be lost in the process of rebooting.

★ Windows 9x and ME do not support deferred data writes



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Disabling Forced Writes on a Linux server

Linux servers are safer for running an operation with forced writes disabled temporarily. Do not leave it disabled once your large batch task is completed, unless you have a very robust fall-back power system.

3. Restoring a backup to a running database

One of the restore options in the `gbak` utility (`gbak -r[estore]`) allows you to restore a `gbak` file over the top of an existing database. It is possible for this style of restore to proceed without warning while users are logged in to the database. Database corruption is almost certain to be the result.

- ❖ Be aware that you will need to design your Admin tools and procedures to prevent any possibility for any user (including SYSDBA) to restore to your active database if any users are logged in.
- ❖ For `gbak` instructions see chapter 21, [Database Backup and Restore](#) (p. 390).

For instructions about blocking access to users, see chapter 14, [Getting exclusive access to a database](#) (p. 283).

If is practicable to do so, it is recommended to restore to spare disk space using the `gbak -c[reate]` option and *test the restored database* using `isql` or your preferred admin tool. If the restored database is good, shut down the server. Make a filesystem copy of the old database and then copy the restored database file (or files) over their existing counterparts.

4. Allowing users to log in during a restore

If you do not block access to users while performing a restore using `gbak -r[estore]` then users may be able to log in and attempt to do operations on data. Corrupted structures will result.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Where to next?

How to get help

The community of willing helpers around Firebird goes a long way back, to many years before the source code for its ancestor, InterBase® 6, was made open source. Collectively, the Firebird community does have all the answers! It even includes some people who have been involved with it since it was a design on a drawing board in a bathroom in Boston.

- Visit the official Firebird Project site at <http://firebirdsql.org> and join the user support lists.
- Visit the Firebird knowledge site at <http://www.ibphoenix.com> to look up a vast collection of information about developing with and using Firebird.
- Read the *Firebird Reference Guide*—[Resources and References](#) (ch. 10 p. 399) for a collection of useful resources about Firebird, SQL and database application development.

Using these books

This book—*Using Firebird*—and its companion volume, *The Firebird Reference Guide*, have been designed for easy use and access during your development work. The button at the top right-hand corner of each “content” page will cause Acrobat Reader to switch back and forth between the two volumes. Each content page also has a navigation bar with buttons to take you directly to the index listings for the selected character. All index listings are hyperlinked to their sources. If the reader is set to display at 125% resolution, one complete page and the hypertext side menu will be visible.

For greater detail about setting up your server and your network, refer to the early chapters of this book. Chapter 7 is a troubleshooting reference. The ensuing chapters deal in turn with design, language and development issues and provided detailed instructions for using the command-line tools.

The Firebird Project

The developers, designers and testers who gave you Firebird and several of the drivers are members of the Firebird open source project at Sourceforge, that amazing virtual community that is home to thousands of open source software teams. The Firebird project's address there is <http://sourceforge.net/projects/firebird>. At that



Symbols **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

site are the source code tree, the bug tracker and a number of technical files which can be downloaded for various purposes related to the development and testing of the codebases.

The Firebird Project developers and testers use an email list forum—firebird-devel@lists.sourceforge.net—as their “virtual laboratory” for communicating with one another about their work on enhancements, bug-fixing and producing new versions of Firebird.

Anyone who is interested in watching progress can join this forum. However, user support questions are a distraction which they do not welcome. **Please do not try to post your user support questions there!**



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

CHAPTER 2

About Firebird

Firebird is a powerful, compact client/server SQL relational database management system which can run on a variety of server and client operating system platforms including Windows, Linux and several other UNIX platforms, FreeBSD and Apple Macintosh OS/X.

It features a higher level of compliance with SQL standards than any other industrial-strength client/server RDBMS on the market today, while implementing some powerful language features in the vendor-specific sphere of procedure programming.

Firebird's origins

Developed as an ongoing open source project, Firebird is the first new-generation descendant of Borland's InterBase 6.0 Open Edition code which was released for open source development in July, 2000, under the InterBase Public License (IPL).

The Firebird source code tree is maintained on the international open source code foundry, Sourceforge, by a large team of professional developers who donate time and expertise voluntarily to fix, develop and enhance this popular and feature-rich database management software.

The Firebird software products are distributed completely free of registration or deployment fees.

Overview of Features

Firebird is a true client/server software, architected for use in local and wide-area networks. Accordingly, its core consists of two main software programs: the database server, which runs on a network host computer, and the client library, through which users on remote workstations connect to and communicate with databases managed by the server.



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

TCP/IP is the network protocol of choice for Firebird on all platforms, although Windows Networking (NetBEUI) is supported for networks having Firebird running on a Windows NT, 2000 or XP host server.

It is possible to run both server and client on the same physical machine and have the single client connect to the server through TCP/IP local loopback. On Windows machines, a local client can also connect to a database by sharing inter-process communications memory with the Firebird server.

Firebird Server

Firebird server software makes efficient use of system resources on the host computer. The server process uses approximately 2MB of memory. Each client connection is likely to add approximately 115KB to server memory consumption, more or less, according to the characteristics of the client applications and the database design.

Disk space for a minimal Firebird installation ranges from 9MB to 12MB, depending on platform. Additional disk space is required for temporary storage during operation and additional memory is needed for database page caching. Both are configurable according to performance demands and the likely volume and type of data to be handled.

Firebird server runs on a variety of platforms, including

- Microsoft Windows NT 4.0 and Windows 2000 (Server or Workstation editions)
- Windows 95/98 and ME
- Windows XP (Home, Professional and .NET editions)
- Linux, FreeBSD and several UNIX operating systems
- MacOS X (Darwin)

Firebird clients

A remote workstation or a local client requires only the shared **gds** library—a dynamic link library on Windows and a shared object on other platforms—and an application program which can pass and receive parameters to and from the library's interface. The size of the **gds** library is approximately 350 Kb.

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

Generally, you would install a copy of the client library on the host server, for use with several of the Firebird command-line utilities and/or any server-based management programs you might use. Many of these utilities can be run remotely, however. A remote system administrator can manage some of the essential services provided by these utilities by accessing them through a host service controller process.

For Java development, Firebird provides the InterClient Java client interface using the JDBC standard for database connectivity. Alternatively, a newer Firebird JDBC/JCA-compliant Java driver provides more flexible Java application interfacing to Firebird databases. Client applications written in Java can run on any client platform that supports Java, even if it is not among the platforms supported by "native" distributions, e.g. Internet appliances with embedded Java capabilities.

Summary of features

Firebird Feature	Description
SQL support	<p>Firebird conforms to entry-level SQL-92 requirements. It supports formal, cascading referential integrity constraints, updatable views, and full, left and right outer joins. Client applications can be written to the Firebird API, a library of messenger functions through which client applications and the server program communicate.</p> <p>The Firebird server supports development of dynamic SQL client applications. It also ships with a host-language precompiler and in-engine language support for embedded SQL development in host languages such as C/C++ and COBOL.</p> <p>Firebird supports several extended SQL features. Some, including stored procedures, triggers, SQL roles, and segmented blob support, anticipate SQL99 extensions to the standard.</p>



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Firebird Feature	Description
Multiuser database access	Firebird is designed to provide for many client applications accessing a single database simultaneously. Client applications can also access multiple databases simultaneously. Firebird will automatically protect cross-database transactions through a two-phase commit mechanism. SQL triggers can notify listening client applications of specific events such as insertions or deletions.
User-defined functions	User-defined functions (UDFs) can be written and stored on the server machine in external shared object libraries. Once it is declared to a Firebird database as an external function, a UDF is available to any client application accessing the database, as if it were a native function of the SQL language. This flexibility accounts for the very small footprint of the server engine: Firebird database application solutions are deployed without the extra cargo of a server that supports hundreds of unused functions natively in its engine.
Transaction management	Firebird provides full and explicit control to client applications for starting, committing, and rolling back transactions. Every transaction exists in its own consistent context determining isolation from other transactions and resolution of multi-user conflicts at commit time. A transaction's uncommitted view of the state of the database is kept consistent with its initial view and any changes which are made within its own context. Client applications can isolate multiple tasks in separate transactions simultaneously. A single transaction can bridge a task involving an unlimited number of connected databases, with an automatic two-phase commit mechanism to protect integrity, should a database become unavailable before the transaction completes.



For more detailed information, see chapter 8, [Transactions in Firebird](#) (p. 90).



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Firebird Feature	Description
Multigenerational architecture	<p>Firebird uses a multi-generational architecture, by which multiple versions of each data row are created and stored as it accessed by a transaction. In a background thread, extinct versions are garbage-collected and the current and pending versions are managed, in order to give each transaction a persistent view and to resolve priorities when update conflicts occur.</p> <p>The multi-generational architecture of Firebird means that readers never block writers. Firebird allows any row to be visible to any transaction, even if other transactions have updates pending for it.</p> <p>The Firebird engine maintains version statistics which it uses, in conjunction with the isolation and lock response attributes of each transaction, to determine which transaction gets priority when conflicting updates are requested.</p>
Optimistic row-level locking	<p>In Firebird, user-initiated locking is unnecessary. The engine locks a row to other transactions only when a transaction signals that it is ready to update it. This is known as <i>optimistic row-level locking</i>. This style of locking has great advantages in increasing throughput and reducing serialization for client tasks, when compared with systems that lock the rows, or even the entire tables, from the moment the transaction begins.</p>
BLOB Filters	<p>Firebird provides the capability for the developer to supply filter code for converting stored BLOBs from one format to another. For example, a BLOB filter could be written to output a text BLOB, stored in RichText format, as XML or HTML; or to output a stored JPEG image in PNG format. The filters, written in the developer's language of choice and compiled for the server platform OS, are stored on the server machine in a shared object library and declared to databases, exactly as UDF libraries are.</p>



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Firebird Feature	Description
Database administration	<p>Firebird provides command-line tools for managing databases and servers. Thanks to its open source character, Firebird is also abundantly supported by open source, freeware and commercial GUI database administration utilities. Using his or her preferred constellation of tools, the database administrator can</p> <ul style="list-style-type: none">• Manage server security• Back up and restore a database• Perform database maintenance• View database and lock manager statistics
Managing security	<p>Firebird maintains a security database (currently hard-coded as isc4.gdb) storing user names and encrypted passwords. It is located in the root directory of the server installation and controls access to the server itself and all databases in its physical domain. The SYSDBA account has full, destructive privileges to all databases on the server.</p> <p>Firebird provides the capability to define ROLES at database level. Within a database, only SYSDBA and the database owner have full privileges; otherwise, all privileges must be granted explicitly to individual users and/or roles. It is possible—and recommended—to define a set of permissions for a role and then grant that role to specific users as required.</p> <p>The SYSDBA can add and delete users' accounts names and modify the details of an account, including the password. Passwords, once stored, are not human-readable, even by the SYSDBA.</p>



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Firebird Feature

Description

Backing up and restoring databases	<p>The command-line tool gbak backs up a database by dismantling it into a compact structure in which metadata, data and current configuration settings are stored separately. The gbak program also performs some important housekeeping tasks on the database itself during the backup process. The file generated is not readable as a database file.</p> <p>A backup can run concurrently with other processes accessing the database because it does not require exclusive access to the database.</p> <p>The same tool is used to restore a backup file into a database. Databases can be restored into a new file or can overwrite an existing database file.</p>
------------------------------------	--

Database backup and restoration can also be used for:

- Erasing obsolete versions of database records
- Changing the database page size
- Changing the database from a single-file to multiple files
- Transferring a database from one operating system to another
- Upgrading any 32-bit InterBase® database that was backed up by its own version of **gbak**
- Backing up only a database's metadata to recreate an empty database.

Several user-friendly GUI front-ends are available for **gbak**, both as stand-alone tools and as utilities within some of the database administration programs. It is also very simple to set up OS-level scripts, batch files or daemons to perform backups.



For information about database backup and recovery, see chapter 21, [Database Backup and Restore](#) (p. 390).



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Firebird Feature	Description
Other tools	<p>Firebird ships with several other command-line administration tools, including</p> <ul style="list-style-type: none">• isql—an SQL query utility which can run dynamic SQL (DSQL) and several specialized statements interactively or in batch from a script. This is the tool to use for quick access to information about your metadata and for running data definition scripts—see chapter 10, Interactive SQL Utility (isql) (p. 152).• gfix—a database housekeeping and repair kit for minor corruptions. This tool is often used in combination with some of the utilities in the gbak program for identifying and recovering damaged data—see chapter 24, Housekeeping & Repair: gfix (p. 469).• gsec—a command-line interface to the security database, isc4.gdb—see chapter 22, Managing Security (p. 414).• gstat—a utility for printing out the current configuration and statistics of a running database—see gstat command-line tool in chapter 23, Server and Database Statistics.• gds_lock_print or lockpr—a utility for printing out the Lock Manager report on a running database. See Lock statistics in chapter 23, Server and Database Statistics.
Other APIs	Firebird provides two additional APIs to assist developers to produce applications for system administration and for installation and deployment.

Comparing Superserver Architecture with Classic

Firebird servers ship with two distinct architectures for managing multiple client connections. On Windows, Firebird uses a multi-client, multi-threaded implementation of the Firebird server process, known as "Superserver architecture", replacing the Classic implementation used for older InterBase(R) predecessors. Both Superserver and Classic architectures are available for Firebird on Linux, Solaris and possibly others.

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

The so-called "Classic" architecture of InterBase(R) server starts a separate process for each connection to a database under its control. Each client is allocated its own database cache buffers. SuperServer serves many clients simultaneously within a single process. Instead of separate server processes for each connection it uses threads of a single process and pools the database cache buffers for use by all connections.

If you are upgrading from a previous version of Firebird or faced with the choice between Classic and Superserver, the information listed in Table 1 (below) will help to explain what the differences are and how they affect database operations.

The server architecture does not affect the structure of databases or the way client applications work. Firebird databases built on a Classic server can be operated on by an equivalent Superserver server, and vice versa. The same gds client library can connect to either server.

In other words, if you begin by installing the Superserver distribution of Firebird on your Linux host machine and, later, decide to change to Classic, any applications you wrote for your Superserver-hosted databases will work unmodified and the databases themselves will continue to behave as they did before.

- ★ At the time of writing, no publicly distributable Classic server software was available for Windows.
- 📖 Additional details about Classic and Superserver server architectures can be found in the technical documentation pages of the IB Phoenix web site, <http://www.ibphoenix.com>.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Comparison of characteristics

TABLE 2–1 Comparison of Classic and SuperServer architectures

Feature	ARCHITECTURE	
	Classic	SuperServer
Executable	<code>gds_inet_server</code>	<code>ibserver</code>
Processes	Multiple, on demand, one instance per client connection	Single server process, each client runs in its own thread
Lock management	<code>gds_lock_mgr</code> process	Implemented as a thread
Local access	Direct I/O to the database file is possible	Uses a network style of access
Resource use	One cache per process	One cache space for all clients
Security on UNIX	Executable and Lock Manager must run as root	Can run as a non-root <code>uid</code> , with access restrictions

Executable and processes

- **Classic** Runs on demand as multiple processes. When a client attempts to connect to a Firebird database, one instance of the `gds_inet_server` executable is initiated and remains dedicated to that client connection for the duration of the connection.
- **SuperServer** Runs as a single process, an invocation of the `ibserver` executable. `ibserver` is started once by the system administrator or by a system boot script. This process runs always, waiting for connection requests. Even when no client is connected to a database on the server, `ibserver` continues to run quietly. The SuperServer process does not depend on `inetd`; it waits for connection requests to the `gds_db` service itself.



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

Lock management

- **Classic** For every client connection a separate server process is started to execute the database engine, and each server process has a dedicated database cache. The server processes contend for access to the database, so a Lock Manager subsystem is required to arbitrate and synchronize concurrent page access among the processes.
- **SuperServer** The lock manager is implemented as a thread in the `ibserver` executable. Therefore Firebird does not use the `gds_lock_mgr` process. The lock management thread uses inter-thread communication mechanisms rather than POSIX signals that are used by `gds_lock_manager` on Classic.

Resource use

- **Classic** Each instance of `gds_inet_server` keeps a cache of database pages in its memory space. While the resource use per client is greater than in SuperServer, Classic uses fewer overall resources when the number of concurrent connections is low.
- **SuperServer** employs one single cache space which is shared by client attachments, allowing more efficient use and management of cache memory when the number of simultaneous connections grows larger.

Local access method

- **Classic** Permits application processes that are running on the same machine as the database and server to perform I/O on database files directly.
- **SuperServer** Supports local access on Windows platforms only. It requires applications to request `ibserver` I/O operations by proxy, using a network method. SuperServer does not support true local access as Classic does, but simulates a network connection in the shared inter-process communication (IPC) space. This is important because Classic's use of direct I/O is faster and safer than the IPC mechanism used by SuperServer.

On non-Windows platforms—and recommended for Windows too—local connections are made through the localhost server (at IP address 127.0.0.1, by convention).



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

Security on Linux and UNIX platforms

- **Classic** server executables `gds_inet_server` and `gds_lock_mngr` must run as root. The lock manager must have the superuser privilege to send signals to the processes.
- **SuperServer** Can be configured to run as a non-root `uid`, for enhanced security. The permissions on database files can be restricted to allow only the Firebird server `uid` to access the database.

Which is better?

In abstract terms, neither architecture is a clear winner. One architecture generally outshines the other under specific workload conditions:

- A single application running on the same machine as the server is faster with the Classic architecture
- For an application embedded in an appliance, Classic is better, because it provides a single process from application to disk.
- On a single-processor machine, an application with larger numbers of frequently contending clients is faster with SuperServer, because of the shared cache.
- On SMP machines, small numbers of clients whose data updates do not impact others' tasks work better in the Classic architecture.

System Requirements

Server Memory (all platforms)

Firebird server process The Firebird server process makes efficient use of the server's resources, utilizing around 2Mb of memory.

Client connections Each connection to the server adds approximately 115K, more or less, according to the style and characteristics of client applications and the design of the database schema.

Database cache The default is configurable, in database pages. Superserver shares a single cache among all connections and increases cache automatically when required. Classic creates an individual cache per connection.



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

Disk space

Server	A minimal server installation requires disk space ranging from 9MB to 12MB, depending on platform.
Client library	350 Kb
Command-line tools	~ 900 Kb
Db management utility	1Mb - 6 Mb, depending on utility selected.

Minimum machine specifications

Operating system	Version	CPU	RAM
Microsoft Windows	NT 4.0 with service pack 6a	486DX2 66MHz min., Pentium 100MHz or greater recommended	<ul style="list-style-type: none">• 16Mb min. for client• 64Mb min. for multi-client server
	Windows 95/98/ME		
	Windows 2000 with service pack 1	Single CPU recommended	
	Windows XP		
	Red Hat 6.2 or higher, TurboLinux 6.0 or higher, SuSe 7.0 or higher, Mandrake 7.2 or higher	Intel 486 or higher	<ul style="list-style-type: none">• 16Mb min. for client• 64Mb min. for multi-client server
Solaris	2.6 or 2.7	SPARC, UltraSPARC	<ul style="list-style-type: none">• 16Mb min. for client• 64Mb min. for multi-client server



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Solaris	?	Intel	<ul style="list-style-type: none">• 32 Mb min.• 64 Mb for multi-client server
Apple Macintosh	Mac OS/X (Darwin)	See distribution notes	See distribution notes
FreeBSD	v.4.x	See distribution notes	See distribution notes
HP-UX	10.0 or higher	See distribution notes	See distribution notes

Document Conventions

General body text is in this font.

Passages in this font are code, scripts or command-line examples.

❖ Passages highlighted like this are to draw your attention to something important that might affect your decision to use the feature under discussion.

★ Passages highlighted like this contain tips, bright ideas or special recommendations.

📖 Passages highlighted like this contain references to other parts of this book, the companion volume *Firebird Reference Guide* or other documents which provide related information.

The Index, Table of Contents and the document tree which optionally displays to the left of the pages all hyperlink to the corresponding section.

At the top of each page is an Index button which will jump to the first page of the classified Index.

Hyperlinks from the body of the book to other topics in the book look like this. Links to the companion volume look like this: *Firebird Reference Guide*—PSQL-Firebird Procedural Language (ch. 3 p. 222).

[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

CHAPTER 3

About Clients and Servers

In this chapter we take a look at the essential pieces of client/server systems as they are implemented in Firebird and examine how applications interact with the client and server.

What is a Firebird client?

A *Firebird client* is an application, typically written in a high-level language such as C, C++, Delphi, Java, PHP or Perl, that provides end-user access to the features and tools of the Firebird database database management system and to data stored in databases. The isql interactive SQL utility is an example of a client application.

In the client/server model, applications never touch the database physically. Any application process converses with the server through the Firebird client library which resides on the client workstation. It surfaces a programming interface of function call structures known as the Firebird API. This client library must be installed on every user's workstation. Generally, other layers are also involved in the interface between the application program and the Firebird client, providing generic or application-language-specific mechanisms for populating and calling the API functions.

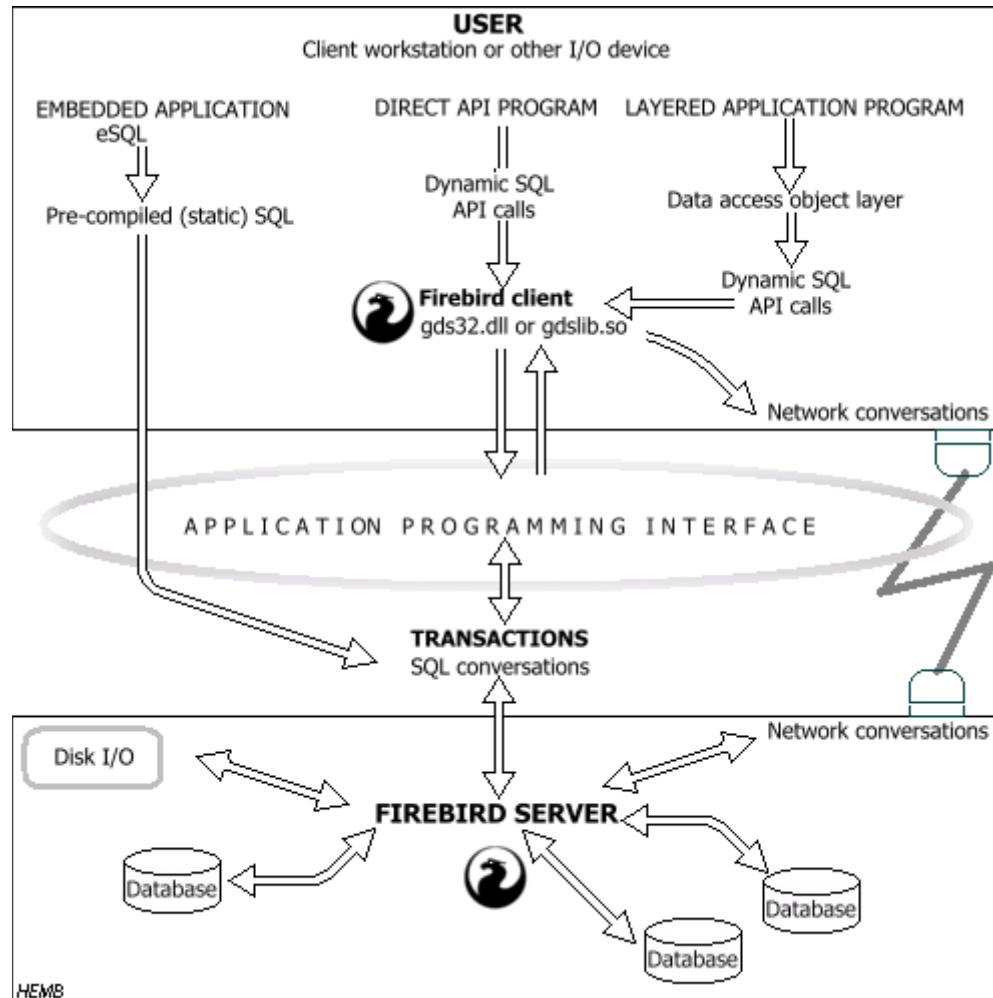
Firebird clients typically reside on remote workstations and connect to a Firebird server running on a host node in a network. Firebird also supports *local connection*, that is, a client application, the Firebird client library and the Firebird server all executing on the same physical box.

Firebird is designed to allow clients to access a server on a platform and operating system different from the client's platform and operating system. A common arrangement is to have Windows 98 or ME workstations concurrently accessing a departmental server running Windows NT or Windows 2000, or any of several flavors of UNIX or Linux.

FIGURE 1 The Firebird client/server model



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z





[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

The Firebird client library

The Firebird client library, currently named **gds32.dll** on Windows and **gdslib.so** on Linux/UNIX, provides functions that developers can use to initiate connections to a server and to perform database operations. The library uses the operating system's client network interface to communicate with one or more Firebird servers, and implements a special Firebird client/server application protocol on top of a network protocol.

The client library provides a set of high-level functions in the form of an Application Program Interface (API) for conversing with a Firebird server. All client applications and middleware must use this API in some way to access Firebird databases. The Firebird API is backwardly compatible with the InterBase API. The *InterBase API Guide* (available from Borland) provides reference documentation and guidelines for using the API to develop high-performance applications. Additional features available in the Firebird API are documented in the Firebird Reference Manual. More recent enhancements are documented in the Firebird release notes.

The server

The Firebird server is a program that runs on a machine with which client workstations can communicate by way of a network. Clients connect to databases physically located on this *server host machine*. The same machine that hosts the executing Firebird server process must host the Firebird databases in its own storage space. The server process is the only process that can perform direct I/O to the database files. It is fully network-enabled, serving multiple connections simultaneously, in response to requests from other nodes in the network. If the network runs under TCP/IP protocol, the scope of the network is virtually limitless.

In the SuperServer architecture, the server process is multi-threaded. In Classic, a new process is started for each connection.

The server's job is to

- regulate access by transactions to individual sets of data
- ensure that each transaction gets and keeps a consistent view of the permanently stored data which it has requested through the client
- receive requests to modify or delete a row and either
 - grant a transaction exclusive write access to the row or

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

- deny access if another transaction already has a write pending.
- maintain the statistics for each database
- maintain and refer to the metadata for each database, in order to manage transactions, data and "house-cleaning".

Clients' requests result in the server performing tasks such as

- creating new databases
- creating new data structures inside databases
- validating and compiling source code for procedures
- searching tables for data matching provided criteria
- collating, sorting and tabulating sets of data
- passing sets of data back to the requesting client
- modifying the values of data
- inserting new data into tables
- removing (deleting) data
- executing compiled procedures
- routing messages to clients

Application development

Once you create and populate a database, its information can be accessed through a client application. Some applications--such as the Firebird isql tool, EMS QuickDesk, IB_SQL, IBAccess or IBConsole--provide the capability to query data interactively and to create new metadata.

Any application developed as a user interface to one or more Firebird databases will use the SQL query language, both to define the sets of data that can be stored and to pass requests to the server about rows it wants to update, insert into or delete from. SQL statements also convey the values which the application wants to be applied to those rows.

Firebird implements a set of SQL syntaxes which have a high degree of compliance with the recognized SQL-92 standards. The Firebird API provides complete structures for packaging SQL statements and the associated parameters and for receiving the results.



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

Embedded Firebird applications

Firebird provides the capability to embed SQL statements in applications written in C or C++, or another programming language. The code is then passed through **gpre**, the pre-processor, which substitutes the embedded SQL statements with code equivalent to that which calls functions in Firebird's client API library. The **gpre** pre-processor generates a file that the host-language compiler can compile.

A special, extra subset of SQL-like source commands is available for this style of application, which are pre-processed into internal macro calls to the API. Known as Embedded SQL (ESQL), it provides a simpler, high-level language syntax for the programmer, that **gpre** can interpret and re-code according to the more complex structure of the equivalent API calls.



The *InterBase Embedded SQL Guide* (available from Borland) provides reference documentation and guidelines for writing embedded Firebird applications.

Pre-defined queries

Some applications are designed with a specific set of requests or tasks in mind. These applications can specify exact SQL statements in the code for preprocessing. The **gpre** Firebird application code preprocessor translates statements at compile time into an internal representation. These statements have a slight speed advantage over dynamic SQL, because they do not need to incur the overhead of parsing and interpreting the SQL syntax at run time.

Dynamic applications

Applications often need to cater for ad hoc SQL statements entered by users at run time. Typically, the application provides a list of columns for the user to select and prompts the user for values to be used as search criteria. The program constructs the query string from the user's selections and inputs.

Firebird uses Dynamic SQL (DSQL) for processing this type of query-on-the fly at run time. Delphi data access components provide properties and methods to analyze and parse request statements and manage the results passed back. Applications that use ODBC or other generic interfaces rely on DSQL statements almost exclusively, even if the application interface provides visual interface tools such as query builders. For



Symbols **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

example, Query By Example (QBE) or Microsoft Query provide convenient dialogs for selecting, restricting and sorting data drawn from a BDE or ODBC data source, respectively.

Component interfaces provide methods and properties for building and preparing SQL template statements, allowing you to use placeholders for value criteria. At run-time, the application supplies *input parameters* of the appropriate data type to complete the statement. Provision is made also for retrieving *output parameters* from statements that return results after the statement is executed.

Component Support for the API

With the rise of rapid application development (RAD) tools in the past decade, the encapsulation of the API functions in suites of components presents a variety of attractive application development options for Firebird developers.

Borland RAD environments

The Borland Database Engine (BDE)

Borland markets "enterprise versions" of a number of integrated development tools--Delphi, Kylix, C++Builder, JBuilder and some older products--which can use the proprietary Borland Database Engine and native SQL Links InterBase drivers as a "black box" middleware layer to make InterBase and, latterly, Firebird databases behave like desktop databases. BDE version 5.2 and its associated InterBase driver, which first shipped with Delphi 6E, supports both Firebird and InterBase version 6, although it has known bugs affecting the Dialect 3 date and time datatypes.

Because the BDE's purpose is to surface a generic, database-vendor-independent, client-centered data access layer to the IDE tools, it flattens out the differences between a wide range of different database systems. Hence, it limits the capability of applications to exploit the best features of Firebird, particularly multiple transactions per connection, control of transaction aging and concurrency control.

The BDE can be useful where you need to develop an application that might be used with a choice of back-ends, of which Firebird is only one.



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

DBExpress™ and Datasnap™

DBExpress and Datasnap were introduced in later versions of the Borland tools to provide alternative generic interfaces to databases. They replace the BDE by moving its functionality into expanded native drivers for supported databases. Like the BDE, they do not support multiple concurrent transactions. They are of especial use where a data interface is required that is independent of the idiosyncrasies of different database management systems.. The InterBase native drivers should provide adequate support for Firebird databases where optimizing client/server performance is not high among the objectives.

Direct-to-API Components

In response to the shortcomings of the BDE, a number of component suites have become available for Delphi and Borland C++Builder that bypass the BDE layer completely and encapsulate the Firebird API directly.

- IB Objects is the most mature, having been in production for four years. It offers two BDE-free suites for data access, one compatible with the native Delphi and C++Builder TDatasource and visual controls, the other completely independent of the Delphi data access architecture and supplied with its own visual controls.
- FIBPlus developed from FreelBComponents suite of Gregory Deatz, offers a connectivity based on TDataset.
- InterBaseXpress (IBX) was also developed from FreelBComponents. Its TDataset-based data access components were purchased for developing as a proprietary product by Borland. Components encapsulating the new Services API were added. It was left unfinished in 1999. Its source code was opened under the InterBase Public License in 2000 and it continues to be developed as an open source project. Borland distributes versions of IBX with some Delphi™, Kylix™ and C++Builder™ versions.



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

Other RAD environments

Microsoft “open connectivity”

Third-party ODBC and OLE-DB drivers are available for Windows programmers using Microsoft and other vendor's rapid application development tools.

Java

A pure Java Type 4 JDBC driver—Jbird or “Jaybird”—is available for Firebird. Jbird is compliant with both the new JCA standard for application server connections to enterprise information systems and the established JDBC standard.

Firebird has abandoned re-implementation of the InterClient and Interserver platform-independent client layers.

API applications

The Firebird client program supports three discrete application programming interface (API) modules. The most important is the core API, through which all database work is performed. Two lesser API modules—the Services API and the Install API—provide functions for accessing various command-line and other utilities from application programs. Developers can write high-level programming or script language applications that populate the structures and call the API functions directly.

The Firebird core API

Programming with the core API requires the application developer to write code for allocating and populating the data structures that provide the communication layer between the client library and the server. Interactive SQL clients, component interfaces and embedded SQL “hide” these structures from the programmer by encapsulating them their own higher level interfaces. Writing code that calls the API functions directly can be more powerful and flexible, with the following benefits:

- Control over memory allocation
- Simplification of compiling procedure—no precompiler
- Access to error messages
- Access to transaction handles and options



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

Core API function categories

API functions can be divided into eight categories, according to the object on which they operate:

- Database attach and detach
- Transaction start, prepare, commit, and rollback
- Statement execution calls
- Blob calls
- Array calls
- Database security
- Information calls
- Date and integer conversions

The Services API

The opened InterBase 6.0 code from which Firebird was developed surfaced for the first time an application programming interface (API) providing a function call interface to certain server activities such as backup/restore, statistics and user management. Many of these calls provide programming interfaces to the code in the command-line tools. A few lower-level server functions are included as well, some of which overlap functions already available in the core API.

Borland's InterBaseXpress (IBX) components include a subset—known as the *Service components*—encapsulating access to services API calls from some versions of their Delphi™, Kylix™ and C++Builder™ development environments.

❖ The Services API and the Borland service components work only with Superserver servers.

The Install API

The Install API provides a library of functions that enable you to develop programs for installing Firebird. For example, it is possible to create a "silent install" that is transparent to the end-user.

★ This API should be regarded as deprecated in Firebird until such time as the source code should be released to open source.



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

Server-side programming

Among Firebird's powerful features for dynamic client/server application programming is its capability to precompile source code on the server, storing the object code right inside the database in most cases. Such procedures and functions are executed completely on the server, optionally returning values or data sets to the client application. Firebird provides three styles of server-side programming capability: stored procedures, triggers and user-defined functions (UDFs).

Stored procedures

Firebird's procedure language (PSQL) implements extensions to its SQL language, providing conditional logic, flow control structures, exception handling (both built-in and user-defined), local variables, an event mechanism and the capability to accept input arguments of almost any type supported by Firebird. It implements a powerful flow control structure for processing cursors which can output a dataset directly to client memory without the need to create temporary tables. Such procedures are called from the client with a SELECT statement and are known to developers as *selectable stored procedures*.

Stored procedures can embed other stored procedures and can be recursive. All stored procedure execution, including selection of data sets from procedures and embedded calls to other procedures, is under the control of the single transaction that calls it. Accordingly, the work of a stored procedure call will be cancelled totally if the client rolls back the transaction.

Triggers

Triggers are special procedures created for specific tables, for automatic execution during the process of committing DML work to the server. Any table can have any number of triggers to be executed before or after inserts, updates and deletions. Execution order is determined by a position parameter in the trigger's declaration. Triggers have some language extensions not available to regular stored procedures or to dynamic sql, notably the context variables OLD and NEW which, when prefixed to a column identifier, provide references to the existing and requested new values of the column. Triggers can embed other stored procedures.

Work performed by triggers will be rolled back if the transaction that prompted them is rolled back.



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

Stored procedures and triggers can not start transactions.

User-defined functions

By design, in order to preserve its small footprint, Firebird comes with a very modest arsenal of internally-defined (native) data transformation functions. Developers can write their own very precise functions in familiar host-language code such as C/C++, Pascal or Object Pascal to accept arguments and return a single result. Once an external function—UDF—is declared to a database, it becomes available as a valid SQL function to applications, stored procedures and triggers.

Firebird supplies two libraries of ready-to-use UDFs: ib_udf (available for both Windows and Linux) and FBUDF, currently available for Windows and under development for Linux. Firebird looks for UDFs in libraries stored in the /udf directory of its installation or in other directories configured by the `external_function_directory` parameter in the Firebird configuration file.

Multi-database applications

Unlike many relational databases, Firebird applications can be connected to more than one database simultaneously. The Firebird client can open and access any number of databases at the same time. Tables from separate databases can not be joined to return linked sets, but cursors can be used to combine information.

If consistency across database boundaries is required, Firebird can manage output sets from querying multiple databases inside a single transaction. Firebird implements automatic two-phase commit when data changes occur, to ensure that changes cannot be committed in one database if changes in another database, within the same transaction context, are rolled back or lost through a network failure.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

CHAPTER 4

Operating Basics for Firebird Server

This chapter provides a short introduction to the basic tasks and commands for operating the Firebird server.



OTHER REFERENCES

For information about configuring the server, see chapter 5, [Configuring the Firebird Server](#) (p. 60)
For information about configuring network clients and connecting to Firebird, see chapter 6,
[Network Configuration](#) (p. 72)

Running Firebird on Linux/UNIX

The Firebird server process **ibserver** runs as a daemon on Linux/UNIX. The command-line utility **ibmgr** is used to start and stop the process.

Starting the server

To start the Firebird server, log in as the "*root*" or "*firebird*" user.

- ❖ Once you have started ibserver using one login, such as "*root*," be aware that all objects created belong to that login. They are not accessible to you if you later start **ibserver** as one of the other special users.
We strongly suggest you run the Firebird server under the "*firebird*" login.
- For Firebird **SuperServer**, execute the following from a command shell:

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

`ibmgr -start -forever`

- Firebird **Classic** uses the `xinetd` or `inetd` process, depending on Linux version, to handle incoming requests. There is no need to explicitly start the server. `xinetd` or `inetd` forks off a process to handle incoming requests. Usually, `xinetd` or `inetd` is set up to start automatically.

Stopping the server

For safety, make sure all databases have been disconnected before you stop the server. Firebird does not provide a utility for counting users connected to a database on this platform. If you need to allow clients a grace period to complete work and detach gracefully, shut down individual databases using the `gfix` tool.

You do not need to log on as "root" to stop the Firebird server. Execute the following command:

```
ibmgr -shut -password SYSDBA_password
```

The `-shut` option rolls back all current transactions and shuts down the server immediately.

Other `ibmgr` commands

Syntax

From the command shell:

```
ibmgr -command [-option [parameter] ...]
```

Alternatively, you can start an *interactive ibmgr session*, i.e. go into "prompt mode".

Type:

```
ibmgr [Enter]
```

to bring up the `ibmgr` prompt.

In prompt mode, the syntax for commands is:



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

IBMGR> command [-option [parameter]]

The command switches **-user** and **-password** can be used as option switches for commands like **-start** or **-shut**. For example, you can shut down a server in either of the following ways:

From the command line:

ibmgr -shut -password password

In prompt mode:

IBMGR> shut -shut -password *password*

ibmgr commands and switches

start

start [-once | -forever]

Starts server; the **-forever** switch causes the server to restart if it crashes; default is **-forever**

shut

Rolls back current transactions, terminates client connections, and shuts down server immediately

show

Shows host and user

user *user_name*

Supplies SYSDBA

password *password*

Supplies SYSDBA password

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z****help**

Prints help text

quit

Quits prompt mode

Making the server start automatically

To configure a Linux/UNIX server to start the Firebird server automatically when the host server boots up, install a script that the rc initialization scripts can run.

Refer to */etc/init.d/README* for more details on how Linux/UNIX runs scripts at boot time.

The Firebird 1.0 install of SuperServer should handle the automatic start of the server on bootup. The appropriate script should be installed by default. Of course, testing this across all the different releases of the different distributions is difficult. However, Firebird 1.0 on SuSE 7.3 and RH 7.2 installs seamlessly, with the server configured to start on bootup. Even for distros where the install fails to set things up correctly, the start/stop script will have been installed in */etc/init.d*.

Sample start/stop script

Below is a sample script that can be run to start or stop the Firebird daemon on Linux. Other sample scripts can be found in the *Firebird Reference Manual*.

```
#!/bin/sh
# ibserver.sh script - Start/stop the Firebird daemon
# Set these environment variables if and only if they are not set.
: ${INTERBASE:=/usr/firebird}
: ${ISC_USER:=SYSDBA}
: ${ISC_PASSWORD:=masterkey}
# WARNING: in a real-world installation, you should not put the
# SYSDBA password in a publicly-readable file. To protect it:
# chmod 700 ibserver.sh; chown root ibserver.sh
export INTERBASE
```

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

```
export ISC_USER
export ISC_PASSWORD
ibserver_start() {
# This example assumes the Firebird server is
# being started as UNIX user 'firebird'.
echo '$INTERBASE/bin/ibmgr -start -forever' | su firebird
}
ibserver_stop() {
# No need to su, since $ISC_USER and $ISC_PASSWORD validate us.
$INTERBASE/bin/ibmgr -stop
}
case $1 in
'start') ibserver_start ;;
'start_msg') echo 'Firebird Server starting...\c' ;;
'stop') ibserver_stop ;;
'stop_msg') echo 'Firebird Server stopping...\c' ;;
*) echo 'Usage: $0 { start | stop }' ; exit 1 ;;
esac
exit 0
```

Running Firebird on Windows

On Windows platforms, the Firebird server program **ibserver.exe** is usually monitored by the Guardian program **ibguard.exe**, which will attempt to restart **ibserver.exe** if it terminates abnormally. The **ibserver.exe** program can also run without Guardian protection. Guardian is regarded as redundant on Windows NT, 2000 and XP and recommended on Windows 95/98 and ME.

- On Windows NT and Windows 2000, Guardian and the Firebird server program can run as services or as applications. The default installation installs Guardian and the Firebird server to run automatically as services. When Firebird runs as a service, a small degree of administration, including stopping and restarting, can be done through the Firebird Manager control panel applet.

Guardian and Firebird server can be changed to run instead as applications on Windows NT and Windows 2000.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- On Windows 95/98, ME and XP, Firebird can run only as applications.

When Firebird runs as an application, an icon appears in the system tray. Certain administration tasks can be done manually by right-clicking on the tray icon.

Running as a service - NT, 2000 & XP

When Firebird was started as a service, use the Firebird Manager control panel applet to start and stop the Guardian and Firebird server.

Firebird Server Manager

Start the Firebird Server Manager control panel applet by selecting Start | Settings | Control Panel | Firebird Manager

(the appearance of the installed applet may differ slightly):



Use Firebird Manager if you need to do the following:



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- Change the **server startup mode**.
 - **Automatic** sets it to start up automatically at system boot-up
 - **Manual** lets you start it manually or from an application.
- The change does not take effect until next time the server machine is rebooted.
- Change the **path to the root directory** of the Firebird server. Use this if you have relocated the Firebird files to a different directory. Click **Change**, browse to the new directory for the Firebird root and select it.
 - ❖ As a general rule, you should keep the relative paths the same.
- Change Firebird server to **run as an application** next time it starts up. Unless you have a special contrary requirement, it is *strongly recommended* to keep Firebird server running as a service.
 - ❖ Now that CPU affinity on SMP servers is a configuration parameter in the **ibconfig** file, there is no longer any need to run Firebird as an application in order to set the CPU affinity by way of the **IB_Affinity** program.

Running as an application on any Windows platform

If Firebird server is running as an application, you should see an icon in the system tray of the server machine. Right-click on the tray icon to display a GUI program providing the ability to shut down the server, along with some limited administrative capability.

Starting Firebird as an application manually

If Firebird server is not running, it can be started or restarted manually by selecting it from the **Start|Programs|Firebird** menu.

Alternatively, Guardian can be started from the command prompt:

- Invoke the command prompt ("MS-DOS") window
- Change to the \bin folder of your Firebird installation.
- Start the Guardian with the following command:



Symbols [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

`ibguard -a`

When Guardian starts up , it automatically starts ibserver.exe.

You can start the server directly yourself, without the protection of Guardian, by using this command instead:

`ibserver -a`

The server will start as an application and place its own icon in the System Tray.

Stopping the server

If Firebird is running as a service on Windows NT, 2000 or XP:

- Open the Firebird Manager applet from the control panel and click **Stop**. Guardian stops the server and then shuts itself down. The Firebird Manager applet continues running.

If Firebird is running as an application on any Windows 32-bit version:

- Right-click the Firebird Guardian icon and choose **Shutdown**. Guardian stops the server and then shuts itself down.

Configuration

As would be expected of a highly scalable software system, Firebird server has a large set of configurable parameters. A default set of values is applied to the server at installation time. It is highly unlikely that the default configuration will be suitable for all, or even most, requirements.

- Read the chapter [Configuring the Firebird Server](#) to determine what to expect from the defaults and how to manipulate the settings to suit the needs of your users and their applications.

Brief introduction to the command-line tools

Firebird comes with a number of command-line tools for doing administrative tasks. In general, they work the same way in both the Linux/UNIX and MSDOS shells. On Linux/UNIX, case-sensitivity of commands, parameters and switches is a consideration. On Windows, it is not an issue.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
isql

Interactive query utility with tools for managing transactions, displaying metadata, creating, editing, retrieving and executing scripts containing batches of SQL statements.

For details about using **isql**, refer to the chapter [Interactive SQL Utility \(isql\)](#).

gbak

For backing up and restoring databases. Because it operates at the structural and data format levels, this is the only right utility to use for safe backups. It also detects corruption, frees disk space left tied up by deletions and resolves uncompleted transactions.

- ❖ A backup-and-restore procedure is necessary when splitting your database into multiple files and also enables you to transport a database from one operating system to another.
- ❖ Never use file-copy backup utilities like gzip, Winzip, MSBackup, filesystem copying or third-party utilities for backing up or transporting a database unless the server is completely shut down and you are certain the database is free of corruption. Databases that are transported as file copies will contain uncollected garbage.

For details about using **gbak**, refer to the chapter [Database Backup and Restore](#).

gsec

Command-line interface to the isc4.gdb security database, for managing user accounts on the Firebird server.

For details about using **gsec**, refer to the section [Maintaining the security database](#) in the chapter [Managing Security](#).

gfix

This is a set of housekeeping utilities for reconfiguring database properties, doing minor repairs, performing various cleanup tasks, etc. It also provides the means for the administrator to shut down individual databases before a server shutdown.

**Symbols** [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

For details about using **gfix**, refer to the chapter [Housekeeping & Repair: gfix](#).

gstat

Extracts and displays index and data statistics for a database.



For details about using **gstat**, refer to the section [gstat command-line tool](#) in [Server and Database Statistics](#).

gds_lock_print and iblockpr

Utility to retrieve statistics from the lock file that Firebird maintains to control the consistency of database changes by multiple transactions. On Linux/UNIX the program is named **gds_lock_print**; on Windows it is **iblockpr.exe**. It can be a useful analysis tool when deadlocks are a problem.



For details about using **gds_lock_print** or **iblockpr**, refer to the section [Lock statistics](#) in the chapter [Server and Database Statistics](#).

ibmgr

For starting and stopping the Firebird server on Linux and UNIX host platforms. Details are at the start of this chapter.

[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

CHAPTER 5

Configuring the Firebird Server

This chapter describes how to configure the environment variables and server parameters on the machine on which the Firebird server is installed.

Environment variables

Environment variables are system-wide settings that take effect when the operating system is booted. On Windows, Linux and most Unix systems, the Firebird server will recognize and use certain environment variables if they are configured. The **ibserver** process (SuperServer architecture) or the **gds_inet_server** process (Classic architecture) will not recognize settings that point to resources not physically controlled by the server machine.

For example, any environment variable that points to a networked resource will not be recognized.

Where to set environment variables

- On Windows NT/2000, define environment variables as *User variables for Administrator* in the **Control Panel | System | Environment** dialog.
- On Linux and UNIX, the easiest way to define environment variables is to add their definitions to the system-wide default shell profile. The root user can also issue **setenv()** commands from a shell.

ISC_USER and ISC_PASSWORD

If you do not provide a user name and password when you connect to a database or when you run utilities such as **gbak**, **gstat**, and **gfix**, Firebird looks to see if the **ISC_USER** and **ISC_PASSWORD** environment variables are set; if they are, Firebird uses that user and password as the Firebird user.

❖ Although setting these environment variables is convenient, omit them if security is at all an issue.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

INTERBASE

The **INTERBASE** environment variable is used both during installation and during runtime.

During installation, it defines the path to the root directory where Firebird is installed. The Value must point to a fully qualified path that exists.

If this path is different from `/usr/firebird` (on Linux/UNIX platforms) or `C:\Program Files\Firebird` (on Windows platforms), then all users must be able to find the correct path by reading the **INTERBASE** variable.

INTERBASE_TMP

The **INTERBASE_TMP** environment variable can be used to set the location of InterBase's sort files on the server. The Value must point to a fully qualified path that exists.

There are other options for defining the location of these files. See the topic describing configuration of [tmp_directory](#).

INTERBASE_LOCK and INTERBASE_MSG

INTERBASE_LOCK sets the location of the InterBase lock file and **INTERBASE_MSG** sets the location of the Firebird message file. These two variables are independent of each other and can be set to different locations. The Value must point to a fully qualified path that exists.

TMP

On the server machine the TMP environment variable points to a directory path on the server where Firebird should store temporary files. If the **INTERBASE_TMP** environment variable is not defined and no other sort space has been set up in the Firebird configuration, Firebird will attempt to store temporary sort files here.

Additionally, *on a client machine*, the TMP setting is the only place to control the space where the command history files created by an ISQL session will be stored. If the TMP location is not set, a Firebird client uses whatever temporary directory it finds defined for the local system. If no temporary directory is defined, it uses `/tmp` on a Linux/UNIX client or `C:\temp` on a Windows client.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

The Firebird Configuration File

Configuration parameters for the Firebird server are specified by entering their names and values in the Firebird configuration file, a plain text file which is located in the installation root.

- On Linux/UNIX the configuration file is named **isc_config**
- On Windows, it is named **ibconfig**.

Entries are in the form:

parameter value

- *parameter* is a string that contains no whitespace and names a property of the server being configured
- *value* is a number or string that specifies the value of the property

One parameter and its value occupy one line. Each line is limited to 80 characters, including the word "parameter" and any whitespace characters.

★ The installation process installs a version of the configuration file containing entries for only those parameters which have default settings. All lines are commented out by the '#' symbol. To activate any parameter with a setting different to the default, uncomment the line and edit it to the required value.

Parameters

connection_timeout

Number of seconds to wait before abandoning an attempt to connect. Default 180.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

cpu_affinity

With Firebird SuperServer on Windows, there is a problem with Windows continually swapping the server process back and forth between processors on SMP machines. This ruins performance. Until now, to set ibserver's affinity to a single CPU, it was necessary to run the server as an application and to run a utility (IB_Affinity.exe) on top of the running server program.

This new configuration parameter can be added to ibconfig to remove the need for any external program to change CPU affinity on an SMP Windows system.

CPU_AFFINITY takes one integer, the CPU mask.

Example:

CPU_AFFINITY 1

only runs on the first CPU (CPU 0).

CPU_AFFINITY 2

only runs on the second CPU (CPU 1).

CPU_AFFINITY 3

runs on both first and second CPU.

★ This setting will take effect when the service starts up.

❖ This parameter has no effect in Windows9x or ME, as it uses an NT API call. W9x flavors DO NOT take advantage of multiple processors.

database_cache_pages

Server-wide default number of database pages per database to allocate in memory. The configured value can be overridden by clients.

For more information, see the topic [Configuring the database cache](#) in this chapter.

Defaults vary according to platform and architecture:

Windows: 2048; Linux/UNIX: 2048 (SuperServer), 75 (Classic)



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

deadlock_timeout

Number of seconds before failure to secure a lock will cause a scan to check for a deadlock condition. Default 10.

dummy_packet_interval

Number of seconds to wait on a silent client connection before the server sends dummy packets to request acknowledgment. Default 60.

external_function_directory

This parameter (available only in Superserver) can be used to specify an arbitrary number of locations for user-defined function libraries and/or character set definitions. If this configuration parameter does not exist, Firebird checks the sub-directories ..\udf or ..\intl beneath the root path pointed to by the INTERBASE environment variable.

Examples:

```
external_function_directory      <double-quoted directory path>
external_function_directory      "opt/firebird/my_functions"
external_function_directory      "d:\udkdir"
```

external_file_directory

This new setting for locating EXTERNAL FILES is specific to platform and is implemented only for Windows initially. There is no limit to the number of directories that can be in the search list. Make one entry per directory as follows:

```
external_file_directory      <double-quoted directory path>
external_file_directory      "d:\x-files"
```

lock_acquire_spins

Number of spins during a busy wait on the lock table mutex. Relevant only on SMP machines. Default 0.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

lock_hash_slots

Used for tuning the lock hash list. More hash slots means shorter hash chains. Not necessary except under very high load. Prime number values are recommended. Default 101.

TCP_NO_NAGLE (for Linux only)

By default, the socket library will minimize physical writes by buffering writes before actually sending the data, using an internal algorithm known as Nagle's Algorithm. It was designed to avoid problems with small packets, called tinygrams, on slow networks.

Disabling the TCP/IP Nagle Algorithm can actually improve speed on slow networks. The presence of this new switch on Linux allows developers to determine, for themselves, the possible pros and con's of using this alternative packet handling approach.

The preset isc_config/ibconfig is to use the Nagle algorithm:

```
#TCP_NO_NAGLE 0
```

To disable it you would uncomment this line and change it to

```
TCP_NO_NAGLE 1
```

server_client_mapping

Size in bytes of one client's portion of the memory-mapped file used for interprocess communication. Meaningful only for local connections on Windows. Default 4096. The allowable range is 1024 to 8192.

You might want to increase this setting when dealing with retrieval of large data sets such as graphic blobs. If you are running Firebird as an application, you can also modify this setting from the dialog that appears when you right-click on the Guardian icon in the system tray. New settings take effect the next time the server is started.

server_priority_class

Priority of Firebird service on Windows NT or Windows 2000. 1 = low priority, 2 = high priority. Relevant on Windows NT/2000 only. Default 1.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

server_working_size_max

Threshold above which the OS is requested to swap out all memory. Relevant on Windows NT and Windows 2000 only. Default is system-determined.

server_working_size_min

Threshold below which the OS is requested *not* to swap out memory. Relevant on Windows NT and Windows 2000 only. Default is system-determined.

tmp_directory

A directory—or list of directories—to use for storing temporary sort files. The Firebird server creates sort files when the size of the internal sort buffer is too small to accommodate the rows involved in a sort operation. Each request (for example, CONNECT or CREATE DATABASE) shares the same list of temporary file directories and creates its own temporary files. Sort files are released when the sort is finished or the request is released.

Default is the value of the **INTERBASE_TMP** environment variable or, in its absence, the **/tmp** directory on Linux/UNIX or **C:\temp** on Windows NT/2000.. Because those locations cannot be configured for size, it is usually prudent to use a list of **tmp_directory** entries to ensure that sufficient sort space will be available under all conditions.

Specify number of bytes available in the directory, and the path to a directory which exists on a physical drive with sufficient spare capacity. There is no restriction on the name used for the directory. You can list multiple entries, one per line, and the spaces do not need to be contiguous or confined to a single storage device.

For example, the following entries constitute a list in a single configuration file:

```
tmp_directory    6000000    "d:\fbtemp"
tmp_directory    12000000   "f:\fbtemp"
tmp_directory    4000000   "w:\aladdin"
```

pathname must be enclosed in double quotes, or the server will ignore this entry. Space will be used according to the order specified. If space runs out in a particular directory, Firebird creates a new temporary file in the

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

next directory from the directory list. If there are no more entries in the directory list, Firebird displays an error message and stops processing the current request.

Parameters not applicable to Firebird

v4_event_memsize—Number of bytes of shared memory allocated for the event manager. Default: 32768.

v4_lock_grant_order—1 means locks are granted first come, first served. 0 means locks are granted as soon as they are available, which can result in lock request starvation. Default: 1.

v4_lock_mem_size—Number of bytes of shared memory allocated for the lock manager. Default: 98304.

v4_lock_sem_count—Number of semaphores for interprocess communication. Applies to Classic architecture only. Default: 32.

v4_lock_signal—Linux/UNIX signal to use for interprocess communication. Applies to Classic architecture only. Default: 16.

v4_solaris_stall_value—Number of seconds a server process waits before retrying for the lock table mutex. Relevant on Solaris only. Default: 60.

Configuring the database cache

Database cache is the amount of memory reserved for each attached database. It is set as a number of blocks, or *buffers*, of memory, each buffer equivalent to the size of one database page. By default, the number of buffers that you configure applies to all databases on the server. Since each database on the server may have a different database page size (PAGE_SIZE) configured, this means that a database with a larger page_size will consume more memory than one with a smaller page_size.

- the figure is set high enough to accommodate the page requirements for all attached databases, overall performance is maximized because all database activity can be handled in physical RAM rather than being swapped to disk.
- It is possible to reserve too many pages. If many databases are running simultaneously, your request might exceed the amount of physical RAM available to the system. The server may become

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

starved of RAM, causing some operations to be swapped to disk as the operating system tries to manage excessive database demands with the needs of other running applications (including itself).

- An application that is insert-intensive needs fewer buffers than one that does a lot of indexed retrieval.
- When the server handles many clients for the same database, using different tables or different parts of table, then it needs more buffers.

Calculating the cache size

To calculate the amount of memory required by a database, multiply the number of pages by the page_size, as defined when the database was created and choose a value proportionately lower. Calculate using a ratio that represents an estimate of the proportion of the total database which is likely to be active during normal usage.

- For a database consisting of one file only, the database_cache_pages can be calculated by dividing the maximum file size allowed by the filesystem, minus 1 byte, by the page size.
- For a multi-file database, calculate the database_cache_pages by totalling up the LENGTH values of the secondary files and adding the STARTING AT value of the first secondary file.

The minimum cache size is 50 pages. There is no maximum, but the total allocation must not exceed system resources. It is not necessary to have a cache that will accommodate the entire database. Under Firebird 1.0 it would lead to serious performance problems, especially on Windows platforms.

Default page allocation is

- **Superserver** 2048 pages for each running database. All users share this common cache pool.
As an indication of how resources can be consumed, a single database running at the default settings for PAGE_SIZE and DATABASE_CACHE_PAGES requires 8MB of memory, but three such databases require 24MB of memory.
- **Classic Server** 75 cache pages per client. Each client is allocated its own cache. The amount of memory required is factored by the number of client attachments to each database, so memory usage is calculated by:

$$\text{PAGE_SIZE} \times \text{DATABASE_CACHE_PAGES} \times \text{number of attachments}$$



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Changing cache size

There are several ways to override the database cache size for a specified database:

- **in the configuration file**

For SuperServer installations, you can configure the default number of pages used for the database caches. Modify the default (2048 pages) by changing the value of DATABASE_CACHE_PAGES in the `isc_config` (Linux/UNIX) or `ibconfig` (Windows) file.

❖ When you change this setting, it applies to every active database on the server.

- **using gfix**

The recommended way is to set the cache at **database level**, using the `gfix` command-line utility with the following switches:

`gfix -buffers n database_name`

where **n** is the number of database pages. This approach permits greater flexibility and reduces the risk that memory is overused, or that database caches are too small.

★ To run `gfix`, you must be either SYSDBA or the owner of the database.

📖 For more information about using `gfix` to do this task, see [Changing database settings](#) on p. 472 of chapter 24.

- **using the Firebird Manager control panel applet** (Windows servers only)

- If you are running Firebird as a service on a Windows on NT, Windows 2000 or XP server, you may override the cache size using the Firebird Manager applet in the Control Panel.
- If you are using a Windows 95, 98 or ME server, or are running Firebird as an application on NT, 2000 or XP, you can change the setting through the pop-up dialog which appears when you right-click on the Firebird or Guardian tray icon.

★ The override value will take effect the next time the server is started.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- **using the isql command-line query tool**

To increase the number of cache pages for the duration of one session of the command-line utility **isql**, you have two options:

Either:

- include the number of pages (*n*) as a switch when starting **isql**

isql -c n database_name

n is the number of cache pages to be used as the default for the session; *n* overrides any values set by **DATABASE_CACHE_PAGES** or **gfix** and must be greater than 9.

or

- include *n* as an argument to the connect statement once **isql** is running:

isql > connect database_name CACHE n

A CONNECT statement entered in an **isql** query accepts the argument **CACHE n**. The value *n* can be any positive integer number of database pages. If a database cache already exists because of another attachment to the database, the cache size is increased only if *n* is greater than current cache size.

- **using the database parameter buffer (DPB)**

In applications, the cache size can be set in a database parameter buffer (DPB) using either the **isc_num_buffers** or the **isc_dpb_set_page_buffers** parameter, according to your server's requirements.

- *isc_dpb_num_buffers* sets the number of buffers to be used for the current connection. It makes most sense in a Classic architecture, where each connection has its own buffer pool. In Superserver, it will set the number of buffers to be used for the specific database, if that database is not already open, but will not persist after the server closes the database.
- *isc_dpb_set_page_buffers* is useful in both Classic and Superserver. It sets a default number of page buffers to be used with the database whenever it is opened.

Verifying cache size

To verify the size of the database cache currently in use, execute the following commands in **isql**:



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

```
ISQL> CONNECT database_name;
ISQL> SET STATS ON;
ISQL> COMMIT;
Current memory = 415768
Delta memory = -2048
Max memory = 419840
Elapsed time = 0.03 sec
Buffers = 2048
Reads = 0
Writes 2
Fetches = 2
ISQL> QUIT;
```

The empty COMMIT command prompts **isql** to display information about memory and buffer usage. The "Buffers" line specifies the size of the cache for that database.

[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

CHAPTER 6

Network Configuration

This chapter provides details for configuring Firebird server and clients in a networked client/server environment. Topics discussed include network protocols supported by Firebird, remote connection specifiers, system-specific configuration notes and some network testing tips.

Network protocols

TCP/IP

Firebird supports TCP/IP for all combinations of client and server platforms.

NetBEUI

Firebird supports the Microsoft NetBEUI protocol for Windows NT/2000.XP servers and Windows clients.

Local access

For Windows clients running the Firebird server on the same physical machine. Firebird supports a *local* connection mode involving interprocess communication to simulate a network connection without a physical network interface.

For all clients accessing a local server on any supported platform using TCP/IP, connection can be made through the special localhost server at IP address 127.0.0.1. Even on a Windows machine, a localhost TCP/IP connection is preferred, particular where the user may require multiple connections and/or Firebird's server events notification is implemented in applications.

Mixed platforms

Firebird's design enables clients running one operating system to access a Firebird server that is running on a different platform and operating system to the client. For example, a common arrangement is to have several

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

inexpensive Windows 95/98 PCs acting as client workstations concurrently accessing a departmental server running on Windows NT/2000/XP, Linux, or any of several brands of UNIX.

★ When setting up a Firebird server for TCP/IP networking, it is recommended that you configure the host name files on clients. Whilst it is usually possible nowadays to use the IP address in your connection string in place of the host name, connection through a host name is usually faster. On some Windows clients, older versions of Winsock do not support an IP address in the connection string at all.

Instructions follow (below) for mapping IP addresses to server names.

★ Besides the Windows and Linux platforms, server implementations of Firebird (Classic, Superserver, or both) are available also for Mac OSX (Darwin), FreeBSD, Sun Solaris (Intel and Sparc), HP-UX and potentially AIX, and can be built for a number of additional UNIX platforms.

❖ Firebird does not currently have platform support for any version of Netware by Novell, nor for networking under IPX/SPX protocol.

Connecting to a Microsoft Windows Server

Local connection

String for Windows local connection

To connect to a database, specify a full, physical path (not a mapped or shared drive path) and the database file name. For example, on Windows:

c:\databases\phoenix.gdb



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

Caveats for local connections

The “local server” concept on Windows was implemented originally for the Classic architecture of an earlier InterBase® ancestor to enable developers, particularly those using the Borland integrated development products, to run the database server and the IDE on a single development machine. A local client could use direct I/O to access the database file. It was not intended for production deployment.

When the Superserver architecture was introduced, a workaround was done to make local connections behave like network connections. The workaround involves a single instance of the client program sharing inter-process communication space with the server program. It works well where a single instance of the client program makes a single connection to each database. It starts to come apart when client threads need to be handled.

- ❖ Avoid this form of local connection for user applications if any of the following applies:
 - there is a requirement for the user to make multiple connections to a single database
 - multiple threading of any application processes needs to be implemented, e.g. an ISAPI module in a Windows web application
 - other users require simultaneous remote access to the same database
 - applications need to implement server Events notification

Refer to the TCP/IP notes below for recommendations regarding local connections using TCP/IP.

Windows Networking (NetBEUI)

A Firebird server can run on any 32-bit version of Microsoft Windows. However, on an all-Windows network, only Windows NT, 2000 and XP can provide server hosting for Windows Networking (NetBEUI) clients. On NT Workstation, Windows 2000 Professional and Windows XP Home editions, concurrent client connection limits under Windows Networking are enforced by the operating system software.

Windows 95/98 and Windows ME can not act as NetBEUI hosts.

- ❖ It is not possible for a Firebird client to connect to a database on a Windows host which it sees as a mapped drive or shared resource.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Connection string format for clients on a NetBEUI network

\Server_name\path\Database_name

where \Server_name is the properly-identified node name on the Windows network.

For example,

\FBSERVER\c:\databases\phoenix.gdb

TCP/IP on Windows

A Windows 95 server or client requires a Windows 95 Plus upgrade to provide Winsock 2 support for the TCP/IP protocol. All other 32-bit Windows versions provide the necessary support "out of the box". Concurrent TCP/IP client connections are not subject to operating system limits.

Connection string format for clients on a Windows TCP/IP network

For a remote client connection, the connection string includes the name of the server machine.

Server_name:drive:\path\Database_name

For example,

FBSERVER:c:\databases\phoenix.gdb

TCP/IP local connection

For a thread-safe connection to a local server, use the TCP/IP *reserved local loopback server*. This emulated network host node is identified as **localhost** and has the IP address 127.0.0.1.

For example,

localhost:c:\databases\phoenix.gdb



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Configuring server names and addresses on Windows

If your TCP/IP network is not running a domain name service, it will be necessary to inform each node individually of the mappings of IP addresses to host names in your network. To enable this, update the HOSTS file in the Windows system directory of *each* node (server and client).

- on Windows NT/2000/XP the HOSTS file is located in `c:\winnt\system32\drivers\etc\`
- on Windows 95/98/ME it is located in `c:\windows`

★ If HOSTS is not present, a file named `Hosts.SAM` will be found in the same location. Copy this file and rename it to HOSTS.

Open and edit the HOSTS file with a plain text editor. Server host entries have this format:

```
192.12.13.1      SERVER_NAME      #Firebird Server  
127.0.0.1        localhost        #Local loopback host
```

- The IP address must be the valid one configured for the host in your network.
- Server_name can be any unique name in your network.
- The comments at the right are optional.
- The format is identical, regardless of whether the host is running on Windows or Linux/UNIX.

Inconsistent strings on Windows

Windows will not raise an error if you omit the first backslash character from the connection path. On InterBase® servers and in early betas of Firebird server, multiple users logging in with inconsistent strings could cause unrecoverable corruption of databases unless the bad path syntax were intercepted by application software at connect time. Corruption was caused by Windows wrongly signalling to the server that connections were being made to separate databases.

The bug is fixed in Firebird, so that only the first user connection to a database is permitted to open the file. If it happens that this first connection is made using the "bad" path string, others using the "good" string will be blocked from connecting. The onus is on application developers and the system administrator to ensure that all clients connect using a consistent path string.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Connecting to a Linux or UNIX Server

TCP/IP is the only network protocol available for connecting clients to a Firebird server on Linux/UNIX.

Connection string format for clients to a Linux/UNIX server

Server_name : /path/Database_name

e.g.

FBSERVER : /usr/databases/phoenix.gdb

- ★ Either forward slashes (/) or backslashes (\) are valid as directory separators in the TCP/IP connection strings. Firebird automatically converts either type of slash to the type appropriate for the server operating system.
- ❖ The correct slash character is required in C or C++ code, when you need to use double-backslashes in string literals.
- ❖ If any element of the database file specification contains spaces, the entire string must be enclosed in single quotes.

TCP/IP local loopback connection

Local connection to a Linux/UNIX database can be done *only* through a local loopback.

localhost : /usr/databases/phoenix.gdb

The local machine has an IP address of 127.0.0.1 with a subnet mask of 255.0.0.0

This special IP address has a predefined server name **localhost**, although it is also possible to associate the address with any server name that is otherwise unrecognized in your network.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Configuring server names and addresses

A host name file can be maintained on client machines for mapping server names to IP addresses. It is a plain text file which you can edit with any plain text editor. The file is named **hosts** and it will be found in the **./etc/** directory.

Examples of host file entries are:

```
10.0.0.2      fbserver      # Firebird server
127.0.0.1      localhost     # local loopback server
209.113.237.205 ibphoenix.com # comment is optional
```

Configuring the gds_db service on client and server

The **services** file on both server and client must have correct entries to indicate the port number associated with the named service **gds_db**. The Firebird kits should make this entry automatically for you during the installation but, if the entry or the file is lost through some unforeseen cause, you can add it using a plain text editor.

The file:

- On Windows NT/2000/XP, this file is **C:\windows\system32\drivers\etc\services**.
- On Windows 95/98/ME, this file is **C:\windows\services**.
- On Linux/UNIX this file is **/etc/services**.

The entry is:

```
gds_db 3050/tcp # Firebird Server
```

★ In releases of Firebird subsequent to v.1, there will be a slight change to the current convention.

Port 3050 will be allocated specifically for "compatibility mode" connection for client applications that need to connect to a database with the number 6 in the version string. Another port will be designated for "native Firebird" connection.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- ❖ In a UNIX environment with NIS, the NIS server can be configured to supply the services file to all NIS clients on UNIX workstations.

Configuring the Linux/UNIX inetd daemon

- With Firebird Classic server, the **inetd** or **xinetd** daemon is configured to listen on the **gds_db** service (port 3050). The installation script should make the appropriate entry in the configuration file—**/etc/inetd.conf** or **/etc/xinetd.conf**. If clients are unable to connect to your Classic server, check that the file does have an entry for **gds_db**, using a plain text editor to add it, if necessary.
 - With Firebird SuperServer, the **ibuserver** process takes over the task of listening on the port, and there will be a conflict if both **inetd** (or **xinetd**) and **ibuserver** attempt to listen on the port, causing connection requests to fail.
- ❖ If your SuperServer has this double configuration, use a plain text editor to remove the line in this configuration file that mentions the **gds_db** service.

After any changes, restart **inetd** or **xinetd** with `kill -HUP` to make sure the daemon will use the new configuration.

Testing the connection

Subnet restrictions

- ❖ NetBEUI cannot route network traffic between subnets. TCP/IP can be configured to restrict traffic between subnets. If the client and server are on a complex network with multiple subnets, check whether the network configuration allows you to route network traffic between the client and server in question.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Pinging the server

You can quickly test whether a TCP/IP client can reach the server, using the **ping** command at the command line console. Usage is:

```
ping server_name
```

If the connection is good and everything is properly configured, you should see something like this:

```
Pinging fbserver [10.10.0.2] with 32 bytes of data
```

```
reply from 10.10.0.2: bytes=32 time<10ms TTL=128
```

If you get something like

```
Bad IP address fbserver
```

then your host name file entry for the server_name (in this example, `fbserver`, may be missing or wrongly spelt. For example, all identifiers on Linux/UNIX are case sensitive.

If you see

```
Request timed out
```

it means that the IP address referred to in your host name file cannot be found in the subnet. This may be because there is a typo in the host name file entry or it may mean that the server is not listening, possibly because there is a physical problem on the network.

Check that

- the network is plugged in
- the network wires are not damaged
- the client and server software is properly configured.



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

Firewalls

Even if the host name is resolved successfully, your connection test might fail if the database server is behind a software or hardware firewall that blocks port 3050, preventing the client from reaching the server.

- Ask your network administrator to open port 3050.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

CHAPTER 7

Troubleshooting Connections

In this section you will find some guidelines and tips for troubleshooting and correcting your connections to Firebird databases.

Can you connect to a database at all?

Attempt a local connection

This test applies only to a Windows machine which has a full Firebird server and client installation. To confirm that the Firebird server process is running on the server and able to attach to your database, try a local database connection:

❖ Local connection mode is not available on Linux or UNIX servers.

- 1 Log in to the console of the database server host, and run an application such as command-line **isql** or a Windows GUI management tool such as IB_SQL or IBAccess.
- 2 Attempt to connect to a database without specifying a hostname: list just the path, e.g.
c:\databases\phoenix.gdb

If it works, then the server is running and the database path is good.

Can you connect to a database on local loopback?

You can simulate a client connecting to the server through the remote client interface by connecting in **local loopback** mode.

❖ The host name file (hosts) will need to configure a server name (e.g. localhost) at IP address 127.0.0.1 for this mode of connection. Please refer to [Configuring server names and addresses on Windows](#) and [Configuring server names and addresses](#) if you need instructions for doing this.



CHAPTER 7 Troubleshooting Connections } Is the server listening on the Firebird port?

Symbols **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

- 1 Log in to the console of the database server host, and run an application such as command-line `isql` or a GUI management tool such as `IB_SQL` or `IBAccess`.
 - 2 Attempt to connect to the database using the local loopback path specification, e.g.
 - Windows: `localhost:c:\databases\phoenix.gdb`.
You MUST include a physical drive designator.
 - Linux/UNIX: `localhost:/usr/databases/phoenix.gdb`
Linux/UNIX systems do not use drive designators but all server, path and file names are case sensitive.
- ❖ If local or local loopback connection fails and you are sure that the database name is spelt correctly, then something is wrong with the configuration of the server or the network. Read back through the chapters on server and network configuration and double-check your settings.

Is the server listening on the Firebird port?

The `gds_db` service (port 3050) will not answer if the `ibserver` process on the server has not been started.

Attempt to start the server

- on a Linux/UNIX server, use `ibmgr -start`
- on Windows NT/2000/XP, if Firebird was set up to run as a service, use the Firebird Manager applet on the Control Panel
- on Windows 95/98, ME or on NT/2000/XP where Firebird was set up to run as an application, start Firebird server from the Start Menu or a command window.



If you are not familiar with starting the Firebird server, see chapter 4, [Starting the server](#) (p. 50) for more instructions.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Connection rejected errors

If the client reaches the server host and the **gds_db** service answers but still cannot attach to a database, you might see a “connection rejected” error.

Is the database on a physically local drive?

A database file must not reside on an NFS filesystem, a mapped drive or a share. When the **ibserver** process finds such a case, it denies the connection.

To correct this situation, move your database to a filesystem on a hard disk that is physically local to the database server and correct your connection string to reflect this.

Are the user and password valid?

The client application must use a user and password combination that matches an entry in the security database **isc4.gdb**. This database must be located the Firebird installation root and it must be writable by the **ibserver** process.

User names and passwords apply to the server, not to any specific database. Make sure you are using a valid user and password for that server. For example, if you developed the database on a development server and moved it to a production server, you will need to update user names and passwords on the new server.

Does the server have permissions on the database file?

The **ibserver** process must have operating system level permission to read and write the database file. Check the permissions on the database file, and the **uid** of the **ibserver** process.

- ★ On Linux/UNIX, you have the option (recommended) of running ibserver as user *firebird* or user *interbase*, which are non-superuser **uids**.

Does the server have permissions to create files in the Firebird install directory?

The **ibserver** process must have write permission in the Firebird directory (by default, **/opt/interbase** on Linux/UNIX, **C:\Program Files\ Firebird** on Windows).



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

It must also be able to write to, and perhaps create, the `interbase.log` file (in the Firebird root directory) and other temporary files (in the configured or default locations, see [The Firebird Configuration File](#) on p. 62 of chapter 5).

Can the client find the host?

The error message "Unable to complete network request to host" appears when the Firebird client cannot establish a network connection to the server host. Some of the common causes are:

Wrong BDE reference: If you are using the BDE, you must specify the full Firebird connect string in the `SERVER` property when you set up the alias in the BDE Administrator. In BDE applications, you must use the `AliasName` property for your `TDatabase` connection, NOT the `Path` property.

Wrong BDE or InterBase driver version: the BDE and driver versions that shipped with Borland Delphi 5 were for InterBase 5.x. Whilst they may be able to connect to Firebird, they are not certified as reliable with a Dialect 1 database. Because of data type changes, they imply will not support Dialect 3 at all and possibly will not even connect to a Dialect 3 database.

If you need to use the BDE, upgrade to BDE version 5.2 or higher. BDE 5.2 ships with Borland Delphi 6 Enterprise and Professional Editions. The accompanying InterBase driver unfortunately has reported bugs with handling Dialect 3 date and time types.

★ If you are developing with Borland Delphi or C++Builder, consider using one of the BDE-free connectivity solutions. Links to these products are listed in *Firebird Reference Guide—Resources and References* (ch. 10 p. 399).

IP address used in place of Server Name: If you supplied an IP address instead of a host name (server name) the Firebird client may be unable to resolve it or it may time out while trying to do so. The Firebird client attempts to translate the server portion of your connect string to an IP address, by calling `gethostbyname()`. Some current TCP/IP drivers—including Winsock 2 and Linux TCP/IP—do resolve strings that look like IP addresses but certain older drivers may not have this capability.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Bad or missing gds_db entry in the services file: Not likely to be an issue with Firebird 1.0, since the client no longer needs to look up the Firebird network service by name—it will find port 3050 by default. If an older client gives the network error, you might try creating the `gds_db` entry in the `services` file manually. Alternatively, upgrade Firebird and avoid the problem.



see [Configuring the gds_db service on client and server](#) on page 78 in chapter 6

The client cannot find the host: The server you specify must be running on the network that you use. If the hostname corresponds to a host that is inaccessible because of network interruption, or the host is not running, then the connection request fails with the network error.

The indicated network protocol is not available: The syntax of the Firebird connect string determines the network protocol the client uses to connect to the server host. If your server does not support the protocol indicated by your connect string, the connection attempt fails with the network error.

For example,

- the NetBEUI connection syntax (`\server\c:\path\database.gdb`) works only if your server is an NT server. The syntax does not work if your server is running UNIX, Linux or a version of Windows that cannot act as a NetBEUI host (i.e. Windows 95/98 or ME)
- The Firebird client library does not support IPX/SPX network protocol. Connection will fail if you attempt to use IPX/SPX by specifying the database connect string in the form
`server@volume:/path/database.gdb`.

Firebird server is not running or not installed: A network connection request cannot succeed if the Firebird server is not installed and running on the server host. If there is no process listening for connection requests, the client's connection request fails with the network error.



Symbols [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

Path string is inconsistent with an existing connection: Firebird will block connection if the path string submitted is inconsistent with that used by an existing connection. Firebird added this mechanism to protect databases from a long-standing bug in the antecedent InterBase® code which is known to cause severe corruption to Windows-served databases.



see [Inconsistent strings on Windows](#) on page 76 in chapter 6

Disabling automatic Internet dialup on Windows systems

Microsoft Windows operating systems provide by default a networking feature that is convenient for users who use a modem to connect to the Internet: any TCP/IP request that occurs on the system activates an automatic modem dialing program. It can be a problem on client systems that use TCP/IP to access a Firebird server on a local network. As soon as the client requests the TCP/IP service, the Windows automatic modem dialer fires up, interfering with network connections from client to server.

There are several ways to suppress the automatic modem dial feature. No more than one of these methods should be necessary to configure your system to work as you need it to.

Reorder network adapter bindings

You probably have a dialup adapter and an Ethernet adapter for your local network. On Windows NT/2000/XP , you can reverse the bindings order of the two adapters to force the Ethernet adapter to service the TCP/IP request before the dialup adapter tries. Do this in **Settings | Control Panel | Networking | Bindings | All Adapters | Move Down**

The local ethernet adapter satisfies any local TCP/IP requests and passes on any remaining requests—such as Internet requests—to the next adapter in the list, the dialup adapter.

Use Internet Explorer configuration

If you have Microsoft Internet Explorer installed (as if you could avoid it), you have an item on the Control Panel that allows you to disable the autodial feature of the dialup network driver.

Run **Settings | Control Panel | Internet | Connection tab.**

- On Windows 95, deselect “Connect to the Internet as Needed.”

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

- On Windows NT or Windows 98, check "Connect to the Internet using a local area network." You can also change this in some versions of Internet Explorer. Use the menu **View | Internet Options | Connection** and check "Connect to the Internet using a local area network."

After making this change, you must invoke your modem dialer manually each time you want to use the Internet.

❖ If you don't have Internet Explorer, you won't have any access to these settings.

Disable autodial in the Registry

To disable autodial, do the following:

- 3 Start the registry editor with **regedit.exe**
- 4 Move to the registry key

`HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Internet Settings`

- 5 Find `EnableAutoDial` in the right panel
- 6 Change its data value from 1 to 0

Disable RAS autodial

The easiest way to do this is to disable the RAS AutoDial service:

- 1 Start the services control panel applet **Control Panel | Services**
- 2 Scroll down to "Remote Access AutoDial Manager" and select it
- 3 Click **Startup** and change the startup to Manual; click **OK**
- 4 If you want to stop it now click **Stop**
- 5 Click **Close**

To re-enable the RAS autodial service, repeat the procedure, changing the startup to Automatic.



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

Prevent RAS from dialing out

Perform the following if you are using Windows NT RAS (Remote Access Service):

- 1 Start the Registry editor, using **regedit.exe**
- 2 Move to the Registry key `HKEY_CURRENT_USER\Software\Microsoft\RAS\Autodial\Addresses`
 - ★ A better way to view these is to type **rasautou -s |more** from the command prompt
- 3 In the subkeys look for the server's local address and/or name; select the key and select **Delete** from the **Edit** menu
- 4 Close the Registry editor
- 5 Reboot the machine

You might also wish to add local network addresses to the disabled list:

- 1 Start the registry editor using **regedt32.exe**, instead of **regedit.exe**
- 2 Move to the registry key `HKEY_CURRENT_USER\Software\Microsoft\RAS\Autodial\Control`
- 3 Double click **Disabled Addresses** and add the address on a new line; click **OK** when you are finished
- 4 Close the registry editor
- 5 Reboot the machine

[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

CHAPTER 8

Transactions in Firebird

In a client/server database such as Firebird, client applications never touch the data that are physically stored in the pages of the database. Instead, the client applications conduct conversations with the database management system—"the server"—by issuing requests and handling responses packaged inside transactions. This topic visits some of the key concepts and issues of transaction management in Firebird.

Context of a transaction

Each conversation between a client and the server has a unique *context* that isolates it from all other conversations. Such a conversation involves one or more client requests and one or more server responses. A complete conversation between a client and the server is known as a *transaction*.

Every conversation between a client and the Firebird server involves a transaction, even if the client accesses a read-only database or views a read-only data set.

★ The sole exception to this rule is the call made to a Firebird unique value generator.

📖 For more information about generators, see [Firebird Generators](#) on page 296 in chapter 15 and the *Firebird Reference Guide*—[CREATE GENERATOR](#) (ch. 2 p. 90).

A Firebird transaction is always started by a client. It does not finish until the client asks for it to finish. A client can complete a transaction in one of only two possible ways:

- when the client issues a request to COMMIT it (make any data changes permanent)
- when the client issues a request perform a ROLLBACK (undo any changes that have been requested since the transaction began).



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

★ If no data or metadata changes are pending, a COMMIT request is no different to a ROLLBACK request. Under such conditions, COMMIT is regarded as more efficient than ROLLBACK because the steps to update record versions are fewer.

Failed COMMITS

If changes are pending and an exception occurs, a request to COMMIT the transaction will fail. A transaction is never rolled back by the server engine. It remains the responsibility of the client application to cause the transaction to finish by requesting ROLLBACK.

Rollback never fails

Rollback never fails. It will undo any changes that were requested during the transaction: the change that caused the exception as well as any that would have succeeded had the exception not occurred.

Why does Firebird use transactions?

A data management system that is being updated by multiple users concurrently is vulnerable to a number of data integrity problems that arise when their work is allowed to overlap without any control. In summary, they are:

- *Lost updates*, which occur when two users update a single set of data simultaneously. In reality, both users have the same view of the set and one performs changes, closely followed by the other, who overwrites the first user's work.
- *Dirty reads*, which occur if one user can select changes (her own or others') that are neither committed nor rolled back.
- *Non-reproducible reads*, which occur when one user repeatedly selects rows that other users are updating and deleting. Whether this is a problem depends on the circumstances. For example, a month-end financials process or a snapshot inventory audit will be skewed under these conditions, whereas a travel booking application must keep all users' views continually synchronized.

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

- *Phantom rows*, which arise when one user can select some but not all new rows written by another user. Again, this may be acceptable in some situations but will skew the outcomes of some processes.
- *Interleaved transactions*. These arise from dependencies between rows within a table or between tables that are not adequately protected or enforced by triggers or integrity constraints. They are usually timing-related, occurring when multiple users simultaneously operate on the same data with no predictable sequence or frequency.

Firebird solves all of these problems by means of a transaction management model that isolates tasks inside a unique context—known as *transaction isolation*—that prevents undesired overlapping of task outcomes while maintaining a high level of concurrent access to all users.

★ Firebird does not permit dirty reads. In certain conditions, by design, it allows non-reproducible reads.

The single-transaction model

A client application can issue many requests inside a single Firebird transaction. This has obvious advantages for tasks that need to execute a single statement iteratively with a variety of input parameters; or for tasks where a group of statements must be executed and, finally, committed as a single job or rolled back entirely if an exception occurs.

Generic application interfaces, such as ODBC, the Borland Database Engine (BDE) and Borland's DBExpress, utilize this capability to prevent Firebird and other transaction-capable database management systems from running concurrent transactions from a single connection.

Developers who have become accustomed to desktop databases sometimes mistakenly regard it as a benefit to be able to ignore Firebird's transaction management altogether and "keep things simple". As a direct result of the single-transaction limitation, their application architectures will display some or all of the following characteristics:

- strict serialization of a user's work

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

- “multiple tasking” for a single user implemented as numerous program modules, each isolated by separate connections to the database
- modal work flows
- complicated and confusing multiple-window user interfaces
- long-running transactions around multi-user networks, causing frequent update conflicts, build-up of garbage in the database and regular server hang-ups or crashes
- large data sets with concommittant overloading of the network
- avoidance of SQL through heavy use of table-based data access classes and visual controls or widgets
- heavy use of client-side code to process data
- detachment of data sets into local disk or memory cache to remove them from transaction control

Multiple transactions

A Firebird client can run multiple concurrent transactions. A user working on multiple tasks in a single application can perform a variety of activities over the same (or overlapping) sets of output data. Firebird's transaction model provides great benefits for designs that need to cater for a modular, multiple-tasking environment in a highly responsive manner. It is an important responsibility for the software engineer to develop techniques for keeping commits flowing and synchronizing the views presented to the multi-tasking application user.

Cross-Database Transactions

Firebird supports operations across multiple databases under the control of a single transaction. It implements two-phase commit automatically, to ensure that the transaction will not commit the work in one database unless it is possible to commit it in the other. Data are never left partly updated.

In the first phase of a two-phase commit, Firebird prepares for committing the work for each database, by splitting the transaction into *sub-transactions*, one for each database. To each database it writes the changes pertaining to its respective sub-transaction. In the second phase, Firebird marks each sub-transaction as committed, in the order in which the parts were prepared.

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

Cross-database transactions can use a lot of server resources. For embedded SQL (ESQL) Firebird provides an optional transaction parameter clause which can be used to limit the databases that the transaction can access. The USING <list of dbHandles> clause will specify which databases the transaction is permitted to access.

- ❖ A transaction employing a USING clause cannot request exclusive access to specific tables by means of a RESERVING clause.

Limbo transactions

If network interruption or a disk crash makes one or more databases unavailable, causing the two-phase commit to fail during the second phase, some sub-transactions are flagged as committed and others are not. These *limbo transactions* are left in a state whereby the server does not know whether to commit them or roll them back. Sometimes, records in a database become inaccessible because of their association with a transaction that is "in limbo".

RECOVERY

Until a limbo transaction is finished (by being committed or rolled back) it remains "interesting" to Firebird, which keeps statistics on unfinished transactions. *Recovering* a limbo transaction means committing it or rolling it back. The `gfix` tool can recover limbo transactions.

- 📖 For details of using `gfix` to recover limbo transactions, .
- ★ Applications built with driver layers which do not support two-phase commit—such as Borland's Database Engine (BDE) and DBExpress™—never exhibit limbo transactions.

Record Versions, Concurrency and Isolation Levels

When the same data are involved simultaneously in tasks being performed by more than one transaction, the transactions are said to be *concurrent*.

- ★ The word *concurrency* is often used loosely to refer to the conditions which are set up in the parameters when the client starts a transaction.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Factors affecting concurrency

Three parameters affect concurrency: **access mode**, **lock resolution** and **isolation level**.

For one isolation level (READ COMMITTED) the current states of the **record versions** are also considered.

Record Versions

When an update request is successfully posted to the server, Firebird creates and caches a copy of the original row image as seen by the transaction—sometimes called a *delta*—and a new version of the row, incorporating the requested changes. These two row images are referred to as *record versions*. The "live" version of the record is locked by Firebird and it becomes inaccessible for updating or deletion by any other transactions.

Until the transaction is eventually committed, it is the newest cached version that the server updates. It does not touch the "live" version again until the Commit occurs.

Such updates, along with new record insertions, generally occur as a direct response to operations requested from within that transaction. However, triggers—both formal cascading referential integrity triggers and user-defined ones—will affect the record versions of related rows in dependent tables.

Once the transaction is committed, the newest cached record version becomes the current version. Other versions, now obsolete, are left behind to await garbage collection. This clean-up of extinct record versions occurs in background, either when the record is next accessed, or when a sweep takes place, or when the database is backed up using the **gbak** tool.

In databases that are not regularly backed up, extinct versions may accumulate over time for rows that are infrequently accessed.

Access Mode

Access mode can be READ WRITE or READ ONLY. A READ WRITE transaction can select, insert, update and delete data. A READ ONLY transaction can only select data.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Lock resolution

can be WAIT or NOWAIT.

WAIT

(the default) causes the transaction to wait until rows locked by a pending transaction are released, before determining whether it can update them. At that point, if the other transaction has posted a higher record version, the waiting transaction will notify that a lock conflict has occurred.

- ❖ When considering using WAIT lock resolution, use it with caution. Refer to the item below, regarding "livelock" conditions.

NO WAIT

causes the transaction to notify a lock conflict immediately, if it finds locks on rows it wants to update. In a reasonably busy multi-user environment, NO WAIT is recommended, in preference to creating bottle-necks of contending, unfinished transactions.

The client application can handle lock conflicts through the use of rollback, timed retries or other appropriate techniques.

- ❖ In situations where a quick turnover can not be guaranteed, it will be best to roll back transactions that cannot be committed because of locking conflicts. It is important to reduce the problem of the system being slowed down and eventually being brought to a halt because Firebird can not advance the OAT (Oldest Active Transaction) and perform background garbage collection.

Transaction Isolation

Firebird provides several levels of transaction isolation to define how consistent a particular transaction's view will be. A transaction can be totally isolated from changes made by other concurrent transactions on data it is viewing or it can be kept up to date with changes committed by other transactions. At one extreme, a transaction can get exclusive access to an entire table whilst, at the other, the uncommitted transaction can "see" the effects of changes committed by other transactions since it started. No Firebird transaction will ever see any *uncommitted* changes pending for other transactions.



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

Isolation level can be READ COMMITTED, SNAPSHOT or SNAPSHOT TABLE STABILITY.

Isolation level	Record Version setting	Description
SNAPSHOT	Not applicable	<p>The transaction works with a non-volatile view of the database that remains unchanged, regardless of any changes committed by other transactions before it completes. It is sometimes referred to as <i>concurrency level</i>. It is a useful level for "historical" tasks like reporting and data export, which would be inaccurate if not performed over a reproducible view of the data.</p> <p>★ SNAPSHOT is the default isolation level for embedded programming, the <code>isql</code> query tool and for Delphi transaction components that use the TDataset data access architecture.</p>
SNAPSHOT	Not applicable	<p>A transaction works with a non-volatile view of the database. It also "earmarks" all of the corresponding live data in its purview until it finishes. Although it does not block concurrent transactions from reading the table's data, it prevents them from writing changes to the table. It is sometimes referred to as <i>consistency level</i> because it is guaranteed to deliver its data in a state which will remain consistent throughout the database as long as the transaction lasts.</p> <p>❖ Because of its potential to lock up the database to other users needing to perform updates, SNAPSHOT TABLE STABILITY must be used with caution. Attend carefully to the size of the affected sets, the effect of joins and table dependencies and the likely duration of the transaction.</p>
TABLE STABILITY		

**Symbols** [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

Isolation level	Record Version setting	Description
READ COMMITTED		<p>A this level, the copy versions are updated not just in response to operations requested from within that transaction, but also in response to changes and inserts that are committed by other transactions. This allows the transaction an up-to-date view of the data upon which it is operating, without needing to commit or roll back and without causing lost update problems.</p> <p>By design, READ COMMITTED isolation allows non-reproducible reads and does not prevent the phenomenon of phantom rows. It is the most useful level for high-volume, real-time data entry operations because it reduces data contention, but it is unsuitable for tasks that need a reproducible view.</p> <p>Record version settings</p> <p>For READ COMMITTED transactions, you can also specify how Firebird will decide that your transaction's view needs to be refreshed:</p>
	RECORD_VERSION	causes the transaction to read the latest committed version of a requested row, even if a more recent pending version exists.
	NO RECORD_VERSION	The default setting. If used in conjunction with WAIT lock resolution, causes the transaction to wait until the latest committed version of the row is also the latest version. When considering using WAIT lock resolution, NO_RECORD_VERSION should be used only with caution. Refer to the item below, regarding "livelock" conditions. If NO RECORD_VERSION is used in conjunction with NO WAIT, the transaction immediately returns a "deadlock" error if the latest committed version is not the most recent version.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Table Reservation

Firebird supports a *pessimistic locking* mode to force full locks on one or more tables for the duration of a transaction. The optional RESERVING <list of tables> clause requests immediate full locks on all committed rows in the listed tables, enabling a transaction to guarantee itself exclusive access at the expense of any transactions that become concurrent with it.

Unlike the normal locking tactic, reservation locks all rows *pessimistically*— it takes effect at the start of the transaction, instead of waiting until the point at which an individual row lock is required.

Table reservation has three main purposes:

- to prevent any deadlocks that might occur through optimistic row locking
- to provide *dependency locking*, i.e. the locking of tables which might be affected by triggers and integrity constraints. Dependency locking, while not normally required in Firebird, can ensure that update conflicts arising from indirect dependency conflicts will be avoided.
- to strengthen the transaction's precedence with regard to one or more specific tables with which it will be involved. For example, a SNAPSHOT transaction that needs sole write access to all rows in a particular table could reserve it, while assuming normal precedence with regard to rows in other tables.

Uses of table reservation

- A SNAPSHOT or READ COMMITTED transaction could use it to make access more restrictive for other concurrent transactions.
- A SNAPSHOT TABLE STABILITY transaction could use it to reduce the possibility of deadlock occurring during a critical process.

This mode is not for everyday use. It may be useful for a task such as freezing inventory tables to run a "book count and valuation snapshot" before a stocktake or other inventory audit.



The RESERVING clause has parameters that determine how much protection is requested for each table:

```
RESERVING <reserving_clause>;
<reserving_clause> = table [, table ...]
[FOR [SHARED | PROTECTED] {READ | WRITE}] [, <reserving_clause>]
```

Each table should appear only once in the entire reserving clause.

Parameters for RESERVING

- FOR SHARED READ is the most liberal reserving condition. It allows any transaction to select from the table or tables named in the sub-clause and any READ WRITE transaction to update them.
- FOR SHARED WRITE allows any transaction to select from the table or tables named in the sub-clause and any SNAPSHOT READ WRITE or READ COMMITTED READ WRITE transaction to update them.
- FOR PROTECTED READ prevents other transactions from updating rows but allows any transaction to select from the table(s).
- FOR PROTECTED WRITE prevents other transactions from updating rows but allows any SNAPSHOT READ WRITE or READ COMMITTED READ WRITE transaction to update them.

❖ A transaction may not use both a USING clause and a RESERVING clause.

Locking and Lock Conflicts

With Firebird, locking is controlled by an interaction between transactions and the records managed by Firebird's versioning engine. Except when a transaction is operating at SNAPSHOT TABLE STABILITY level or with a RESERVING restriction on access by others, locks are applied at row level. The timing of the lock on the row is *optimistic*, that is, a lock is not applied until the moment it is actually required, when the transaction posts its intention to change data in that row.

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

A lock conflict is not an error. It is the intended outcome of Firebird's transaction isolation and record versioning strategy - it protects volatile data from uncontrolled overwriting in a multi-user, multi-tasking application environment.

The strategy works so well that there are really only two conditions which can cause lock conflicts:

- **Condition 1:** one transaction updates or deletes a row and another transaction, which started before the original transaction locked that row, attempts to update or delete the same row. The second transaction either has to roll back its attempt and try again when the first transaction commits, or wait until the first transaction either commits or rolls back.
- **Condition 2:** one transaction has a whole table locked against writes because it has the table isolated in a state of SNAPSHOT TABLE STABILITY or by RESERVING it for protective access and another transaction attempts to update or delete a row or to insert a new row.

What to expect when lock conflicts occur

When concurrent SNAPSHOT and READ COMMITTED transactions attempt to update the same row at the same time, lock conflicts occur. Expect that

- for updates where a single UPDATE statement modifies multiple rows in a table (known as a *searched update*) all updates so far will be undone when ROLLBACK is used to resolve a lock conflict on any single row update. The entire update operation must be retried. The impact of such a condition can be minimized by using the NO RECORD_VERSION option on the READ COMMITTED transaction.
- for updates where rows are retrieved and updated from an output set one-by-one (known as *positioned updates*) only the current update is undone. Upon rollback, the cursor on the set must be closed, then reopened, and updates resumed from the point where the conflict occurred.

What is a deadlock?

A deadlock is just a nickname, borrowed from the sport of wrestling, for the condition where two transactions are contending to update the same row and one transaction does not take any precedence over the other. As

**Symbols** [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

in wrestling, the deadlock can only be resolved if one contender withdraws its hold. One transaction must roll back and let the other commit its changes.

Developers should not dread deadlocks. On the contrary, they are the essence of isolating the work of multiple users in transaction contexts. You should anticipate them and handle them effectively in your client application.

What is a "livelock"?

This is the condition you should dread (and avoid!) When two or more contending transactions want to update the same row and both (or all) are set to WAIT, there is no way out for any of them. Each one will just wait indefinitely for a release that simply cannot occur. Essentially, your client application should never set up a situation where more than one transaction can be in WAIT state pending COMMITS by others involving the same row(s).

Another condition that is sometimes referred to as "livelock" is where one transaction, having backed out of a deadlock situation by rolling back, retries continuously without ever being able to commit work because the contending transaction fails to commit or rollback, leaving an unresolvable locking condition. To avoid this style of livelock, your client applications should attend to "both sides" of this potential trap. On the one side, transactions should be overseen by techniques that force them to commit or roll back work within a tolerable duration. On the other, retry timers in client applications should not be permitted to loop infinitely.

COMMIT with the RETAIN option

(also known as *soft commit* or *commit retaining*)

When a transaction is committed with the RETAIN option, any pending rows are committed and a new transaction is opened, preserving the current cursor status. If the RETAIN option is used on a SNAPSHOT or SNAPSHOT TABLE STABILITY transaction, the new transaction preserves the same snapshot of the data as the original transaction had when it started. Each COMMIT with RETAIN marks a "save point" and, if there is a subsequent ROLLBACK, only the changes since the last save point will be affected.

In a busy environment, the RETAIN option can save time and resources but it has some disadvantages:

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

- A snapshot transaction continues to hold the original snapshot in its view, meaning the user does not see the effects of committed changes from other transactions that were pending at the start of the transaction
- As long as COMMITS with RETAIN continue, garbage collection in the database is suspended, resulting in an overall slowdown in performance and the server eventually coming to a standstill or even crashing.

Programming with Transactions

Many language and development environments can interface with Firebird. It is outside the scope of this manual to describe in detail how to manage Firebird transactions from each one.

For several object-oriented development environments, such as Object Pascal, Borland C++, DBI-Perl, PHP and Java, classes of components exist that encapsulate the Firebird API calls relating to transactions comprehensively.

For a structured high-level host language environment, typically ANSI C, Firebird itself implements a subset of SQL-like statement structures that, along with the static SQL statement structures, can be embedded directly in client code in place of the equivalent API calls.

Transactions and the API

The Firebird Header File

The Firebird header file, ibase.h, located in the \include or /include directory of the installation root, contains the C prototypes for the API functions, along with the structure type and parameter definitions and macros required by various API functions. This file (or a translation specific for your host programming language) should be included in application source code and made accessible in your compiler's search path.

For Pascal coding:

- ibase.h has been translated as ibheader.pas by Borland for use with its data access components.
- it is available as IB_Header.pas for IB Objects.

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

- Search the documentation archives at <http://www.ibphoenix.com> for information about locating translations for other languages.

Setting up a transaction parameter buffer (TPB)

The following section summarizes the syntaxes which these tools use, in one form or another, for calling the transaction-related interface functions surfaced by Firebird's application programming interface (API). Application interfaces such as components and drivers generally take care of TPBs.

A TPB is a byte array containing parameters that describe optional attributes of a transaction. The TPB array must be set up and populated before starting a transaction. Each transaction's TPB may be distinct; or transactions with identical characteristics may share a TPB.

Characteristics are set by including a set of TPB constants in the TPB array (vector). For example, the constant `isc_tpb_concurrency` will make the transaction isolation SNAPSHOT and `isc_tpb_write` will make it READ WRITE.

The first array element in every TPB must be the constant `isc_tpb_version3`, a constant used internally by the Firebird engine.

A null pointer may be passed to the function `isc_start_transaction()` in place of a TPB. In that case, Firebird will apply a default set of attributes, viz. `isc_tpb_concurrency` (SNAPSHOT isolation), `isc_tpb_shared` (shared access mode), `isc_tpb_wait` (WAIT lock resolution), `isc_tpb_write` (READ WRITE access mode).

 The full range of TPB constants is listed in the *API Guide* (API.pdf) of the InterBase® 6 documentation set, available from Borland.

Setting up and initializing transaction handles

Any application must provide one transaction handle (a long pointer) for each transaction it starts.

To establish transaction handles for use, a variable of type `isc_tr_handle` must be declared for each simultaneously active transaction.



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

A transaction handle must be set to zero in order to initialize it before starting a transaction. A transaction will fail to start up if it is passed a non-zero handle.

API functions pertaining to transactions

STARTING A TRANSACTION

Two API function calls are available for starting transactions:

- *isc_start_transaction()* starts a new transaction against a database. It can also be used to start a transaction against multiple databases (up to sixteen) if the application language supports passing a variable number of arguments to a function call.
- *isc_start_multiple()* starts a new transaction against multiple databases. It is used with application languages that do not support passing a variable number of arguments to a function call, or where the number of databases being attached to may vary and/or exceed 16 in number.

RETRIEVING A TRANSACTION ID

Firebird does not provide a server-based mechanism for tracking transaction. It is the responsibility of the application program to store and clean up its own transaction handles—which are meaningless on the server—and test them in the context of client-initiated actions that succeed or fail.

Stored Procedures, Triggers and Transactions

Embedded transactions

In Firebird, transactions are always started and finished by a client. Because stored procedures and triggers execute on the server, they can not start and commit transactions. Thus, Firebird does not support embedded transactions.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Stored Procedures

Stored procedures execute in the context of the transaction that calls them. Work done, including any tasks fulfilled in embedded and recursive calls, will only take effect if everything completes without error and is committed. If the transaction is rolled back, all work is rolled back.

Triggers

Triggers fire inside the context of the DML statement that prompts them. Work done, including any tasks fulfilled in embedded procedure calls, updates, inserts or deletes to other tables or through internal referential integrity or other triggers belonging to any of the tables will be committed in entirety or rolled back in entirety.



For information about writing and using stored procedures and triggers, see [Programming on Firebird Server](#) on page 494 in chapter 25

Transaction “Ageing” and Statistics

When a transaction is started on the server by a client, the server assigns an internal identifier to it. It uses these identifiers to track the states of transactions and make decisions about when to wake up the garbage collection thread and which old record versions to flag for cleaning out. The server remains “interested in” a transaction as long as it remains unresolved—neither committed or rolled back.

Transaction statistics

The command-line tool `gstat`, used with the `-h[eaders]` switch, can reveal some useful statistics about “interesting transactions”, specifically the *oldest interesting transaction* and the *oldest active transaction*.

Following is an extract from the output of `gstat -h[eaders]`:

```
Oldest transaction 75
Oldest active 152
Oldest snapshot 152
Next transaction 153
```

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

- `Oldest transaction` is the oldest interesting, i.e. the oldest transaction that failed to commit. It could be rolled back, active, dead, or limbo.
- `Oldest active` is the oldest transaction marked as active.
- `Oldest snapshot` is the high-water mark for garbage collection.

Transaction “age” and garbage collection

Garbage collection—both the background garbage collection and sweeping—are important to the “hygiene” of a database. As obsolete record versions accumulate, performance becomes degraded.

Background garbage collection

Background garbage collection tries to kick in whenever there is idle time in database activity and the Oldest Snapshot changes. If possible, it will clear out old record versions left behind by transactions in which it is no longer interested. It cleans out all garbage from transactions whose numbers are lower than the Oldest Active Transaction (OAT), or lower than the Oldest [Interesting] Transaction (OIT), if the OIT is lower than the OAT.

Moving the OAT forward

As transactions are committed or rolled back, they become uninteresting and the OAT pointer moves forward to the next transaction that is still active. Thus, these uninteresting transactions become candidates for the next round of garbage collection. In a well-behaving system, the OAT and the OIT both point to the same transaction and their numbers will be close to the Oldest Snapshot.

Any unresolved READ-WRITE transaction is interesting. As long as it remains interesting, no garbage collection will be performed on obsolete record versions left behind by newer transactions, regardless of whether the newer transactions have meanwhile become uninteresting. The longer the OAT remains interesting, the more sluggish the database performance becomes.

Client applications that hold READ-WRITE transactions open (active) for lengthy periods, inhibiting effective garbage collection, commonly characterize a database system that frequently slows down, or even hangs up, without evidence in the log of any misadventure. It is one of the most common characteristics of a database

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

application that has been converted to Firebird from a desktop database such as Paradox or Access without attention being given to prompt resolution of transactions.

- ★ READ-ONLY transactions with the READ COMMITTED isolation level do not inhibit garbage collection.

What the statistics can tell you

Transactions are taking too long

When the gap between the Oldest Active (OAT) and the Oldest Snapshot grows to a level that is somewhat higher than the average number of user tasks concurrently running, it is a sign that you have long-running transactions occurring in your system. Avoiding long-running transactions is one of the best skills you can acquire when learning to write client applications.

Teaching users to complete tasks within a reasonable time—not to go for coffee breaks leaving tasks unfinished, for example—can help. However, good client applications should be designed to avoid depending on users to behave well. Mechanisms that cause transactions to “time out” are effective. If a browsing interface is required, those that use READ-ONLY READ COMMITTED transactions for the browsed data are preferred. If that is not practicable, then ensure that READ-WRITE transactions are committed regularly.

- ❖ Be aware that COMMIT RETAINING (“soft commit”—while handy for maintaining a quickly-refreshed editing interface—will inhibit the forward passage of the OAT. Applications should enforce a periodic “hard commit” by calling COMMIT.

Transactions have died

Sometimes, a transaction may remain “interesting”, even though it is no longer active. Such transactions were never resolved but, for some reason, they became inactive. For example, a client application terminated its connection without completing some transactions; or a cross-database transaction became invalid when the connection to one database got lost, leaving limbo transactions behind.



Symbols **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

You can recognize that such events have occurred when gstat -h tells you that the Oldest Transaction (OIT) is older than the Oldest Active (OAT). If such transactions are ignored, the gap will continue to widen and garbage collection will be crippled.



Cleanup of limbo transactions can be done using the gfix utility. For more information, see [Housekeeping & Repair: gfix](#).

Dead transactions can be removed by performing a backup using gbak. For more information, see [Database Backup and Restore](#).

[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)**CHAPTER 9**

Firebird SQL & Queries

SQL (pronounced "ess cue ell") is a data sub-language for accessing relational database management systems. Its elements, syntax and behavior became standardized under ANSI and ISO in 1986. Firebird's SQL language adheres closely to the published standard, known as SQL-92. The first parts of this chapter introduce the elements and flavors of SQL implemented in Firebird.

User applications can access the data in Firebird tables only one way: by querying them. A query is, in its essence, an SQL sentence consisting of keywords, phrases and clauses in a conventional, English-like language that is understood by both the gds client library and the Firebird server—and, desirably, by the human who composes the sentence!. All metadata and data creation, modification and management tasks, without exception, are done with queries.

With experience and research, you will accumulate an instinct for what works best in SQL for Firebird and for the particular conditions in your database and deployment scenarios. In the later parts of this chapter you will find descriptions and explanations of some of the terms and concepts that you will encounter along the way.



IMPORTANT REFERENCES

- For a detailed summary of the syntax and keyword usages in Firebird SQL, see the alphabetically-listed SQL Statement and Function Reference section of the *Firebird Reference Guide*
- A [Glossary](#) of terms is included at the end of this book.
- If your previous contact with SQL has been minimal, a good book on SQL-92 basics is invaluable. See list in the *Firebird Reference Guide*



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

About the SQL standard

Since its introduction, the SQL standard has undergone three major reviews: SQL-89 (published in 1989), SQL-92 (1992 or thereabouts) and "SQL 3", which is a work in progress and has been published in part as "SQL-99".

The standard SQL query language is non-procedural; that is, it is oriented toward the results of operations rather than toward the means of getting the results. It is intended to be a generic sub-language, to be used alongside a host language, as a specialized mechanism for extracting sets of data predictably from relational databases and for passing instructions about data creation and manipulation into them.

So, while the standard prescribes in great detail how the elements of the language shall be expressed and how the logic of operations is to be interpreted, it makes no rules about how a vendor should implement it. A "conforming" implementation is expected to provide the basic set of features and may include other features which the standard groups at higher levels.

Conformance with the standard is a matter of degree, rather than an absolute. Vendors may freely implement language features that the standard does not prescribe. However, if they implement a prescribed feature with idiosyncratic or custom elements, they cannot claim conformance for it. The greater the preponderance of these idiosyncratic features implemented in an SQL engine, the lower its degree of conformance.

Firebird claims a high degree of conformance with the SQL-92 standard (the ISO/IEC 9075:1992) at entry level. It includes also a number of highly conformant Level 1 features. Although Firebird SQL follows standards closely, there are small differences.

Types of SQL statements

An SQL statement is used to submit a *query* to the database. Standard SQL defines three types of query statement:

- 1 Those that SELECT a set of data consisting of one or more columns and one or more rows involving one or more tables



Symbols **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

- 2 Those that manipulate data, by performing an INSERT, UPDATE or DELETE operation upon rows in a single database table. Such queries, along with SELECT queries, are referred to in Firebird as *data manipulation language*, or DML.
- 3 Those that CREATE, ALTER or DROP metadata objects (also known as schema objects or schema elements). Such queries are referred to in Firebird as *data definition language*, or DDL.

Procedural language features

The standard does not prescribe procedural language features since, in principle, it assumes that general programming tasks will be accomplished using the host language. There is no specification for language constructs to manipulate, calculate or create data programmatically inside the database management system.

Those RDBMS engines which support server-based programming usually provide SQL-like statement formats and syntaxes to extend SQL. Typically, they provide certain essential flow structures such as control blocks, IF..THEN..ELSE, conditional loops and exception handling, along with the capability to handle input, local and output variables and exceptions. Each vendor's implementation freely provides its own variants of these constructs. Typically, such code modules in the database are called *stored procedures*.

Firebird provides them as *procedure language* (sometimes referred to as P/L or PSQL) which programmers use to write the source code for stored procedures and triggers. PSQL is a variant of the embedded SQL language (see below), extended to include flow control, conditional expressions, and error handling. It has the unique ability to output datasets which can be directly accessed by SELECT statements.

Certain SQL constructs, including all DDL (Data Definition Language) statements, are excluded.

Embedded language features

Some SQL database management systems—including Firebird—provide the capability to embed SQL statements directly inside 3GL host programming language modules. The standard provides conceptual algorithms by which embedded application programming is to be accomplished but does not make any implementation rules.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Firebird's embedded application programming capabilities include a subset of SQL-like statements and constructs which can be incorporated into the source code of a program for pre-processing before the code goes to the compiler. The embedded SQL language constructs are known as embedded SQL (ESQL). ESQL statements cannot be generated dynamically.

ESQL is not valid in stored procedures and triggers; procedure language (PSQL) is not valid in embedded SQL. ESQL is used in programs written in traditional languages such as C and COBOL. The preprocessor, `gpre`, converts ESQL statements into host language data structures and calls to the Firebird server.



For more information, see the *Embedded SQL Guide* in the Borland InterBase® 6.x Media Kit or *EmbedSQL.pdf* in the corresponding electronic document set.

Static vs Dynamic SQL

SQL statements which are embedded and precompiled in code are sometimes referred to as *static SQL*. By contrast, statements which are generated by a client program and submitted to the server for execution during run-time are known as *dynamic SQL* (DSQL).

Statements executed by the interactive SQL utility (`isql`), or other interactive desktop utility programs are DSQL, as are those processed through client applications that use the API directly or indirectly (through database access drivers like ODBC, JDBC and the BDE).

Static SQL allows host code to bypass the Firebird API, being pre-compiled to use macro calls to the API structures. Because the whole mechanism is pre-compiled, it can execute faster than dynamic statements which are submitted, parsed and prepared at run-time.

- ❖ The format of certain regular SQL statements varies between the static and dynamic SQL variants, or between the embedded and procedural SQL variants.

Statement terminators

- ESQL and PSQL statements are terminated with semi-colons
- The terminator is omitted in DSQL statements passed through the API structures

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

- DSQL passed through the isql interactive query utility requires terminators, which can be set to any printable character from the first 127-character ASCII subset.

Interactive SQL (isql)

The interactive query tool isql uses DSQL statements, along with a two subsets of extension commands (the SET XXX and SHOW XXX groups) which allow certain settings and schema queries, respectively, to be performed interactively. Certain SET commands can also be included in *data definition scripts* (DDL scripts) for batch execution in isql.



For more information, see [Interactive SQL Utility \(isql\)](#) on page 152 in chapter 10.

SQL dialects

In Firebird 1, each client and database has a *SQL dialect*, an attribute that indicates to a Firebird 1 server how to interpret features and elements which are implemented differently in Borland InterBase® versions earlier than version 6 and the more up-to-date code engine from which Firebird is derived.

Dialect is a transition feature allowing the Firebird engine to recognize, accept and correctly process the older features and elements (Dialect 1), to access these older data for conversion to the new features and elements (Dialect 2) or to apply the full Firebird set of features, elements and rules to converted or newly created databases (Dialect 3).

It is possible to create a new database in Firebird as Dialect 1 or Dialect 3. Since Dialect 2 is intended for converting Dialect 1 databases to Dialect 2, its attribute can be applied only to the client program.



For information about the effects of each dialect, see [Migrating to Firebird](#) on page 617 in chapter 27.



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

Statement list

Firebird provides the following SQL statements.



For a detailed description of each one, see the *Firebird Reference Manual*.

TABLE 9-1 Firebird statement list

ALTER DATABASE	ALTER DOMAIN	ALTER EXCEPTION
ALTER INDEX	ALTER PROCEDURE	ALTER TABLE
ALTER TRIGGER	BASED ON	BEGIN DECLARE SECTION
CLOSE	CLOSE (BLOB)	COMMIT
CONNECT	CREATE DATABASE	CREATE DOMAIN
CREATE EXCEPTION	CREATE GENERATOR	CREATE INDEX
CREATE PROCEDURE	CREATE ROLE	CREATE SHADOW
CREATE TABLE	CREATE TABLE	CREATE VIEW
DECLARE CURSOR	DECLARE CURSOR (BLOB)	DECLARE EXTERNAL FUNCTION
DECLARE FILTER	DECLARE STATEMENT	DECLARE TABLE
DELETE	DESCRIBE	DISCONNECT
DROP DATABASE	DROP DOMAIN	DROP EXCEPTION
DROP EXTERNAL FUNCTION	DROP FILTER	DROP GENERATOR
DROP INDEX	DROP PROCEDURE	DROP ROLE
DROP SHADOW	DROP TABLE	DROP TRIGGER
DROP VIEW	END DECLARE SECTION	EVENT INIT



Symbols [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

[EVENT_WAIT](#)

[EXECUTE](#)

[EXECUTE IMMEDIATE](#)

[EXECUTE PROCEDURE](#)

[FETCH](#)

[FETCH \(BLOB\)](#)

[GRANT](#)

[INSERT](#)

[INSERT CURSOR \(BLOB\)](#)

[OPEN](#)

[OPEN \(BLOB\)](#)

[PREPARE](#)

[RECREATE PROCEDURE](#)

[RECREATE TABLE](#)

[REVOKE](#)

[ROLLBACK](#)

[SELECT](#)

[SET DATABASE](#)

[SET GENERATOR](#)

[SET NAMES](#)

[SET SQL DIALECT](#)

[SET STATISTICS](#)

[SET TRANSACTION](#)

[SHOW SQL DIALECT](#)

[UPDATE](#)

[WHENEVER](#)

[POST_EVENT](#)

[EXTRACT\(\)](#)

Queries

The vehicle of communication between a client application and the server engine is the SQL statement. Statements are passed from the client as queries. When defining metadata for use in a Firebird database, we use a lexicon of standard SQL statements and parameters that provide for the creation of an object by its type and name—or *identifier*—and for specifying and modifying its attributes. Also in this lexicon are statements for removing objects.

Queries using this *Data Definition Language* (DDL) subset of Firebird SQL are reserved for the purpose of metadata definition and thus

- should be carefully controlled if implemented in end-user applications
- cannot be used in stored procedures.

 For more information about DDL queries, refer to the chapters [Databases](#), [Domains and Generators](#), [Tables](#) and [Views](#).



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

The commentary in the rest of this chapter refers to Data Manipulation Language (DML)—the dynamic language of queries that select data from tables, invoke stored procedures or create, modify or destroy data.

An SQL statement is a sentence composed of lexical and syntax elements, expressions and constants. When submitted to the server, it becomes a query. All DML statements target data that are persistently stored in databases as tables. They specify *sets of data* and *operations* to be performed upon those sets. The terms *statement* and *query* are used interchangeably by most developers.

The SELECT query

The SELECT query, the only type which can return a multi-row set of data to the client, is the type most visible in client applications designed for interactive data entry and information retrieval. It has the following general form:

```
SELECT <list of columns>
  FROM <table>
  [<JOIN specification>]
  [WHERE <predicates>]
  [GROUP BY <grouped-column-list>] [ORDER BY <column-list>]
```



For detailed information on specifying SELECT queries, refer to [SELECT](#) in the *Firebird Reference Guide*—[SQL Statement and Function Reference](#) (ch. 2 p. 43).

The INSERT query

The INSERT query is used for adding new rows to a single table. SQL does not permit rows to be inserted to more than one table in a single INSERT statement. The statement has two general forms:

Form 1:

```
INSERT INTO table-name (<list of columns>)
VALUES (<matching list of values>)
```



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Form 2:

```
INSERT INTO <table>(<list of columns>)
SELECT (<matching list of values from another table>)
```

For detailed information on specifying INSERT queries, refer to [INSERT](#) in the *Firebird Reference Guide*—[SQL Statement and Function Reference](#) (ch. 2 p. 43).

The UPDATE query

The UPDATE query is used for modifying values in existing rows of tables. SQL does not allow a single UPDATE statement to modify rows in more than one table. An UPDATE query that modifies only the current row of a cursor is known as a *positioned* update. One which potentially may update multiple rows is known as a *searched* update. The UPDATE statement has the following general form:

```
UPDATE table-name
SET column-name = value [,column-name = value ...]
[WHERE <search predicates> | WHERE CURRENT OF cursor-name]
```

If a WHERE clause is not specified, the update will be performed on every row in the table.

The DELETE query

The DELETE query is used for deleting whole rows from a table. SQL does not allow a single DELETE statement to modify rows in more than one table. A DELETE query that modifies only the current row of a cursor is known as a *positioned* delete. One which potentially may delete multiple rows is known as a *searched* delete. The DELETE statement has the following general form:

```
DELETE FROM table-name
[WHERE <search predicates> | WHERE CURRENT OF cursor-name]
```

If a WHERE clause is not specified, every row in the table will be deleted.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Queries that call stored procedures

Firebird supports two styles of stored procedures: *selectable* and *executable*.

- a **selectable stored procedure** is capable of returning a multiple-row set of data in response to a specialized SELECT statement form, as follows

```
SELECT <list of output columns>
FROM procedure-name [(<list of input values>)]
[WHERE <search predicates>]
[ORDER BY <list drawn from output columns>]
```

- an **executable stored procedure** is one which executes and optionally returns a single-row set of data items. The query for executing such a procedure has the following general format:

```
EXECUTE PROCEDURE procedure-name [(<list of input values>)]
```



For further information about writing and invoking stored procedures, refer to the chapter [Programming on Firebird Server](#) and *Firebird Reference Guide*—[PSQL-Firebird Procedural Language](#) (ch. 3 p. 222).

Sets of data

A DML query defines a logical collection of data items known as a *set*. A set consists of a logical collection of data items arranged in order from left to right in one or more columns.

A query may confine a set's specification to being a single row or it may consist of multiple rows. In a multi-row set, the column arrangement of one row is identical to all others' in the set.



The terms *dataset* and *recordset* are synonymous with *set*. People often casually refer to sets as "queries". They are not synonymous.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

A table is a set

A table is a set whose complete specification can be accessed in the system tables by the database server when the table is referred to by a query from a client. The Firebird server performs its own internal queries on the system tables to extract the metadata it needs in order to execute client queries.

Cardinality and degree

A term you will sometimes meet with respect to sets—including tables—is *cardinality*. Cardinality describes the number of rows in a table or output set. More loosely, one may encounter *cardinality of a row* or *cardinality of a key value*, referring to the position of a row in an ordered output set.

The term used to refer to the number of columns in a set is *degree*. By extension, one may encounter a phrase like “the degree of a column”, meaning its position in the left-to-right order of the columns in the set.

Output sets

A common use of a query statement beginning with the keyword SELECT is to output a set to the client application, for the purpose of displaying data to users, in visual formats that may be tabular or as a single-row screen layout; or for printing in reports. The output set may be in no particular row order, or it can be delivered as a *sorted set*, as specified in an ORDER BY clause.

For example, the following query will output a set of three columns from TABLEA, containing every row that matches the criteria specified in the WHERE clause. The rows will be sorted so that the row with the lowest value in COL1 will appear first:

```
SELECT COL1, COL2, COL3 FROM TABLEA  
WHERE COL3 = 'Mozart'  
ORDER BY COL1
```

- If no WHERE clause is provided, the set will contain every row from TABLEA, not just those in which COL3 has the value ‘Mozart’.
- If all columns of TABLEA are wanted, then the symbol ‘*’ can optionally be used instead of itemizing the columns of the set, e.g.

```
SELECT * FROM TABLEA
```

defines an output set consisting of all columns and all rows from TABLEA.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Input sets

Just as a SELECT query can define a set for an output or cursor operation, so the other DML statements (INSERT, UPDATE and DELETE) define input sets to specify the data upon which the operation is to be performed.

- an INSERT query defines its input set by specifying a table and a left-to-right ordered list of identifiers for columns which are to receive constant values by way of the subsequent VALUES() clause. The VALUES() clause must supply a list of values that corresponds exactly to the input set's order and the data types of its members.

The following example defines an input set consisting of three columns from TABLEB. The constants in the VALUES() clause will be checked to ensure that there are exactly three values and that they are of the correct data types:

```
INSERT INTO TABLEB(COLA, COLB, COLC)
VALUES(99, 'Christmas 2003', '2003-12-25');
```

- an UPDATE query defines its input set using one or more SET clauses—a clause beginning with the keyword SET followed by a columnname = value instruction for each member column of the input set. Member columns are separated by commas. Rows in the set are defined by criteria in a WHERE clause. For example,

```
UPDATE TABLEB
SET COLB = 'Labor Thanksgiving Day (Japan)',
COLC = '2002-23-11'
WHERE ...;
```

- A DELETE query cannot specify columns in its input set—it is implicitly always `*'. The rows in its set are defined in its WHERE clause. For example, the following query will delete every row where COLC (a DATE) is earlier than 13th December, 1999:



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

```
DELETE FROM TABLEB  
WHERE COLC < '1999-12-13';
```

Output sets as input sets

SQL has an important feature that uses an input set that is formed from an output set generated within the same SELECT query—the GROUP BY clause. Instead of the output set from the

```
SELECT <input-set> FROM <table-name> [WHERE...]
```

being passed to the client, it is passed instead to a further stage of processing, where the GROUP BY clause causes the data to be partitioned into one or more nested groups. Each partition typically extracts summaries by way of expressions that aggregate numeric values at one or more levels.

For more information, see [The GROUP BY clause](#) on page 135.

Cursor sets

The SELECT keyword can also define a set which is not output to the client but remains on the server to be operated on by a *server side cursor*. On the server, a cursor is a pointer to a row in a set that is processed row-by-row. As the specified processing on one row completes, the cursor moves on to the next row and repeats the task. This continues until the cursor reaches the end of the cursor set.

Queries that define cursor sets often define an input set as well, since the objective of a cursor operation is to use data from the cursor set and perform a similar task on each row. Typically, the task is to take data items from the current cursor set row and use them to insert new rows into the input set, or to modify or delete rows in the input set.

One example is the type of dynamic query where an INSERT operation defines both the input set and a cursor set selected from another table for the purpose of providing values to the input set. This type of query inserts one row into the input set for each row it encounters in the cursor set. In the following example, an INSERT INTO clause defines an input set for TABLEB and a SELECT clause defines a corresponding cursor set from TABLEA to supply values to the input set:

```
INSERT INTO TABLEB(COLA, COLB, COLC)  
SELECT COL1, COL2, COL3 FROM TABLEA
```

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

Cursor sets are commonly used also for "looping" processes inside stored procedures and embedded applications.

For information about cursor sets—

- in stored procedures—see the *Firebird Reference Guide*—[FOR SELECT...INTO...DO](#) (ch. 3 p. 233) construct
- in embedded SQL—see DECLARE CURSOR in the Statement and Function list (*Firebird Reference Guide*) and related documentation in the Embedded SQL Guide (EmbedSQL.pdf) of the InterBase® 6 documentation set, publ. Borland.

Nested sets

It is possible to define both input and output sets that embed other sets. Some examples follow but the list is not exhaustive.

Sub-selects

A data item for an output set can be defined by the output from a SELECT on another table. The "inner" set is said to be *nested*—sometimes termed a *sub-select* or a *nested sub-select*. The output column should be given a new identifier and, for completeness, can optionally be marked by the AS keyword.

The output returned by the inner query must always be a single column and a single row.

In the following example, COLX from TABLEB is output as a column named DESCRIPTION. If the sub-query finds more than one row meeting the WHERE criterion, the user will see the error "MULTIPLE ROWS IN SINGLETON SELECT".

```
SELECT COL1, COL2, COL3,  
(SELECT COLA FROM TABLEB WHERE COLX='Espagnol')  
AS DESCRIPTION  
FROM TABLEA  
WHERE ... ;
```



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

Correlated sub-selects

When the data item extracted in the nested sub-select needs to be selected on the basis of a linking value in the main query, a *correlated sub-select* is possible. Firebird requires fully-qualified identifiers in correlated sub-selects.

In the next example, the sub-selected column is selected by *correlating* COLX in TABLEB to the current value of COL99 in TABLEA:

```
SELECT a.COL1, a.COL2, a.COL3,  
       (SELECT b.COLA FROM TABLEB b WHERE b.COLX = a.COL99)  
          AS DESCRIPTION  
     FROM TABLEA a  
    WHERE a.OTHERCOL = ... ;
```

Predicates

A predicate is simply an expression that asserts a fact about a value. SQL statements generally test predicates in WHERE clauses and make decisions according to whether the predicates are evaluated as true or false.

★ Strictly speaking, a predicate can be true, false or not proven. For SQL purposes, false and not proven predicates are rolled together and treated as if all were false.

Elements of predicates

A predicate consists of three basic elements: two values and an operator which *predicates* the assertion to be tested.

Operators

All of the operators discussed and summarized later in this chapter (see [SQL operators and expressions](#) on page 143) can be predicate operators. For example, the equivalence operator “=” is used to test for exact matches in this predicate:

```
...WHERE TITLE = 'Star Wars'
```



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Here, the assertion being tested is: "The value in the column named TITLE is equal to the constant string, 'Star Wars' ". When the query engine finds a row where the predicate's assertion is true, it selects that row. For all rows where the assertion is not true, it discards the row and moves on.

Values

The values involved in the predicate can be simple, as in the example above, which makes a simple comparison between a data item and a constant, or they can be extremely complex expressions. As long as it is possible for the expressions being compared to melt down to constant values at the moment the predicate is tested, they will be valid, no matter how complex.

NULL predicates

NULL represents the absence of any value—it represents the *state* of a data item, not its value. Only two predicates are valid for testing for NULL: the keyword IS NULL and its counterpart IS NOT NULL.

A predicate such as

```
EXISTS(COL1 = NULL)
```

will return an error because the equivalence operator is not valid for this predicate. The correct predicate would be

```
EXISTS(COL1 IS NULL)
```

You can also test for NOT NULL:

```
EXISTS(COL1 IS NOT NULL)
```

Any data item in a column which does not have the NOT NULL attribute (see [Specifying NOT NULL](#)) or a default value (see [Specifying column default values](#)) will be stored with a NULL token if no value is ever provided for it in a DML statement.

NULL in expressions

Any expression that tries to perform an evaluation that treats NULL as a value will either fail, because of an invalid operation, or be evaluated as false. For example, a predicate like the following:

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

```
. . WHERE TABLEA.COL1 = TABLEB.COLX
```

will evaluate to false if both data items are NULL when the evaluation is performed, causing the row to be discarded from selection.

An expression like

```
. . WHERE COL1 > NULL
```

will fail because an arithmetic operator is not valid for NULL.

NULL in calculations

In an expression where a column identifier “stands in” for the current value of a data item, a NULL data item in the actual calculation will produce NULL as the result of the calculation, e.g.

```
UPDATE TABLEA
```

```
SET COL4 = COL4 + COL5;
```

will set COL4 to NULL if COL5 is NULL.

- ★ In aggregate functions like SUM() and AVG() and COUNT(<specific_column_name>), rows containing NULL values will be ignored for the aggregation. For AVG(), the numerator will be formed by accumulating the non-null values and the denominator will be the count of the rows containing non-null values.

NULL and user-defined functions

NULL cannot be passed as either input to or output from a traditional InterBase® UDF, because the arguments are passed by reference or by value. Most of the UDF libraries available use the InterBase convention.

The Firebird engine is capable of passing arguments to UDFs *by descriptor*—a mechanism that standardizes arguments on Firebird data types—making it possible to pass NULL as an argument to host code, although not to receive NULL as a return value.

The Firebird UDF library FBUDF.dll, available only for Windows servers at the time of writing, provides a family of functions similar to the NVL() function of Oracle®, permitting a value to be tested for NULL and returned as

**Symbols** [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

a specified non-null value in query output. For information, see the *Firebird Reference Guide*—[FBUDF library](#) (ch. 6 p. 257).

Setting a value to NULL

A data item can be made to be NULL only in a column whose definition has not be specified with the NOT NULL attribute—see [Specifying NOT NULL](#) in [Tables](#).

- in an UPDATE statement the assignment symbol is '=':

```
UPDATE FOO SET COL3 = NULL WHERE COL2 = 4;
```

- in an INSERT statement, pass the keyword NULL in place of a value:

```
INSERT INTO FOO (COL1, COL2, COL3)  
VALUES (1, 2, NULL);
```

- in PSQL (stored procedure language), use the '=' symbol when assigning NULL to a variable and use IS [NOT] NULL in the predicate of an IF test:

```
...  
DECLARE VARIABLE foobar integer;  
...  
IF COL1 IS NOT NULL THEN  
    FOOBAR = NULL;  
...
```

The EXISTS predicate

Often, in stored procedures or queries, you want to know whether there are any rows in a table meeting a certain set of criteria. You are not interested in how many such rows exist, only in determining whether there is at least one. The strategy of performing a count(*) on the set and evaluating any returned value greater than 0 as "true" is costly in Firebird. If performed in the context of a different transaction, it is also unreliable.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

The standard SQL predicate **EXISTS(sub-select)** and counterpart **NOT EXISTS** provide a way to perform the set-existence test very cheaply, from the point of view of resources use. It does not generate an output set but merely courses through the table until it meets a row complying with the conditions predicated in the sub-select. At that point, it exits and returns True. If it finds no compliant row, it returns False.

In the first example, the **EXISTS()** test predicates the conditions for performing an update using a dynamic SQL statement:

```
UPDATE TABLEA  
SET COL6 = 'SOLD'  
WHERE COL1 = 99  
AND EXISTS(SELECT COLB FROM TABLEB WHERE COLB = 99);
```

As with other sub-queries, the sub-query in an **EXISTS()** predicate can be correlated— see [Sub-selects](#) on page 123.

The **IN()** predicate

The **IN()** predicate is similar to **EXISTS()** in that it can predicate on a sub-select, for example:

```
UPDATE TABLEA  
SET COL6 = 'SOLD'  
WHERE COL1 IN (SELECT COLB FROM TABLEB WHERE COLJ > 0);
```

In this case, the **IN()** predicate forms a set of all values for **COLB** in the second table that comply with its **WHERE** clause and compares **COL1** with each. It will perform the update on every row of **TABLEA** whose **COL1** value matches any value in the sub-selected set.

Limitations

IN() is not useful where the sub-select is likely to produce more than a few values. It cannot use indexes and there is an upper limit of 1500 values that the comparison set can contain.

This predicate is of most use in forming a predicate set of values from a control table or from *constants*, for example:

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

```
UPDATE TABLEA  
SET COL6 = 'SOLD'  
WHERE COL1 IN ('A', 'B', 'C', 'D');
```

JOIN Operations

In JOIN operations, two or more related tables are combined by linking them through the columns that relate them, to produce output sets containing columns from any of the participating tables. Joining is one of the most powerful features of a relational database because of its capacity to implement end-user contexts of data in almost limitless ways.

Join operations are applicable to client output sets and, in some cases, to cursors (for Embedded SQL) and to FOR SELECT...DO... loops (in stored procedures). An INSERT, UPDATE or DELETE operation cannot involve a joined set.

Ambiguity in JOIN queries

Firebird will not accept JOIN queries containing column references that do not have full, consistent table qualification. Table qualification can be by *table identifier* or by *table aliasing* but, in most cases, it cannot be a mixture of each.

Qualification by table identifier

For example,

```
SELECT TABLEA.COL1, TABLEA.COL2, TABLEB.COLB, TABLEB.COLC  
FROM TABLEA  
JOIN TABLEB  
ON TABLEA.COL1 = TABLEB.COLA;
```

Qualification by table aliasing

For example,

**Symbols** [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

```
SELECT A.COL1, A.COL2, B.COLB, B.COLC  
FROM TABLEA A,  
JOIN TABLEB B  
ON A.COL1 = B.COLA;
```

SQL-89 JOIN syntax

Under SQL-89, the tables participating in the join are listed as a comma-separated list in the SELECT clause of a SELECT query. The conditions for linking tables are specified in the WHERE clause, although they are not actually selection criteria but *link specifications*. By nature, this syntax can implement only the *inner join*—the output will include rows only if link conditions are met in all of the participating tables. This style of join is sometimes referred to as *exclusive*.

Example

```
SELECT A.COL1, A.COL2, B.COLB, B.COLC  
FROM TABLEA A, TABLEB B  
WHERE A.COL1 = B.COLA;
```

Although this syntax is deprecated in later versions of the SQL standard, it is supported in Firebird. It is not recommended for new work because it is inconsistent with the syntaxes for other styles of joins, making maintenance and self-documentation unnecessarily awkward. Some data access software, including drivers, may not handle the SQL-89 syntax well. It can be anticipated that it will be dropped from the standard at some future time.

- ★ There is no evidence that the more modern SQL-92 “conditional” syntax for inner joins performs better—nor that it performs worse.

SQL-92 (conditional) joins

This is the preferred syntax for inner joins in Firebird, to make code more readable and more consistent with other styles of join supported by SQL-92 and later. It is sometimes referred to as *conditional join syntax*.

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

because the JOIN...ON keywords explicitly distinguish the join *conditions* from search criteria. Not surprisingly, this term "conditional" can be confusing!

Example

```
SELECT A.COL1, A.COL2, B.COLB, B.COLC, C.ACOL  
FROM TABLEA A,  
JOIN TABLEB B  
ON A.COL1 = B.COLA  
JOIN TABLEC C ON B.COLX=C.BCOL;
```

Outer joins

In contrast to the inner joins, an outer join operation outputs rows from participating tables, even if no match is found in some cases. Wherever a complete matching row cannot be formed from the join, the "missing" data items are output as NULL. An older term for an outer joins is an *inclusive join*.

Each outer join has a left and right side, the LEFT being the partial set potentially formed by the columns specified for the table that appears before the JOIN keywords, the RIGHT being the partial set potentially formed by the columns specified for the table that appears after.

FULL OUTER JOIN

If the keywords FULL OUTER JOIN are used, the query will output a set consisting of fully populated columns where matching rows were found, as well as partly-populated rows for each instance where no match was made. Where a left-side set exists and a right-side one does not, the data items pertaining to the right-side table are all NULL. Where a right-side set exists and a left-side one does not, the data items pertaining to the left-side table are all NULL.

★ The keyword FULL is optional when specifying a FULL OUTER JOIN.

LEFT OUTER JOIN

If the keywords LEFT OUTER JOIN are used, the query will output a set consisting of fully populated columns where matching rows were found, as well as partly-populated rows for each instance where a left-side set exists and a right-side one does not: the data items pertaining to the right-side table are all NULL.



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

RIGHT OUTER JOIN

If the keywords RIGHT OUTER JOIN are used, the query will output a set consisting of fully populated columns where matching rows were found, as well as partly-populated rows for each instance where a right-side set exists and a left-side one does not: the data items pertaining to the left-side table are all NULL.

Cross joins

Firebird does not support the old, SQL-89 CROSS JOIN syntax, which produces an output set which is the Cartesian product of two tables.

Natural joins

Firebird does not support the NATURAL JOIN (also known as EQUI-JOIN), which forms sets based on matching columns that share equal values, without explicitly citing the join conditions.

Table aliases

When the name of the table is long or complicated, or there are multiple tables, table aliases are useful (and, in some cases, essential) to clarify statements. The query engine treats a table alias as a synonym of the table it represents and they can be used wherever it is valid to use the table's name.

Example

```
SELECT A.COL1, A.COL2, B.COLB, B.COLC, C.ACOL  
FROM TABLEA A,  
JOIN TABLEB B  
ON A.COL1 = B.COLA  
JOIN TABLEC C ON B.COLX=C.BCOL;
```

Firebird enforces fully-qualified column names in joins. In most cases, it is largely a matter of taste whether you use the real table names or aliases. However,

- in a Dialect 3 database that was created using delimited identifiers combined with lowercase or mixed case object names and/or “illegal characters”, typing queries could get quite exhausting and error-prone if table-aliasing were not available.

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

- when a statement performs a self-referencing (re-entrant) join or a self-referencing correlated sub-select, table-aliasing is inescapable.

For example, this self-referencing join produces a denormalized set from a table which stores Departments in a nested structure:

```
SELECT PARENT.Department_Name as Parent_Department,  
CHILD.Department_Name as Sub_Department  
FROM Department CHILD  
JOIN Department PARENT  
ON CHILD.Parent_DepartmentID = PARENT.DepartmentID;
```

Alias to use

Use any useful string composed of any number of characters that are valid for metadata qualifiers, i.e. alphanumeric characters with ASCII codes in the ranges 35-38, 48-57, 64-90 and 97-122.

Queries that count rows

An entrenched practice exists among some programmers of designing applications that need to perform a row count on output sets. Firebird does not have a quick or reliable way to return the number of rows that will be returned in an output set. Because of its multi-generational architecture, Firebird has no mechanism to "know" the cardinality of rows in persistent tables. If an application must have a row count, it can get an approximation using a SELECT COUNT(*) query.

SELECT COUNT (*) queries

A SELECT statement with the COUNT() function call replacing a column identifier will return the approximate cardinality of a set defined by a WHERE clause. COUNT() takes practically anything as an input argument: a column identifier, a column list, the keyword symbol '*' representing "all columns", or even a constant.

For example, all of the following statements are equivalent, or nearly so. However, SELECT COUNT(<some-column-name>) does NOT include in the count any rows where <some-column-name> is NULL:

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

```
SELECT COUNT (*) FROM ATABLE WHERE COL1 BETWEEN 40 AND 75;  
SELECT COUNT (COL1) FROM ATABLE WHERE COL1 BETWEEN 40 AND 75;  
SELECT COUNT (COL1, COL2, COL3) FROM ATABLE WHERE COL1 BETWEEN 40 AND 75;  
SELECT COUNT 1 FROM ATABLE WHERE COL1 BETWEEN 40 AND 75;  
SELECT COUNT ('Sticky toffee') FROM ATABLE WHERE COL1 BETWEEN 40 AND 75;
```

COUNT(*) is a very costly operation, since it can work only by walking the entire dataset and literally counting each row that is visible to the current transaction as committed. It should be treated as a "rough count", since it will become out-of-date as soon as any other transaction commits work.

Although it is possible to include COUNT(*) as a member of an output set that includes other columns, it is neither sensible nor advisable. It will cause the entire dataset to be walked each time a row is selected for output.

★ The exception is when COUNT(*) is included in an output set being aggregated by a GROUP BY clause. Under those conditions, the count will be performed on the aggregated group in the course of the aggregation process. For example,

```
SELECT COL1, SUM(COL2), COUNT(*) FROM TABLEA  
GROUP BY COL1;
```

❖ Do not use SELECT COUNT(*) as a way to check for the existence of rows meeting some criteria. Instead, use the EXISTS() predicate or, alternatively, the IN() predicate.

Considerations with COUNT()

- The result of COUNT() will never be NULL, because it counts rows. If count has been predicated over an empty set, it returns zero. It can never be negative.
- COUNT(*) in a table counts all rows.
- COUNT(column_name) counts all rows where column_name is not NULL.
- COUNT(DISTINCT column_name) counts only the distribution of distinctly different values in that column. That is, all repetitions of the same value are accounted for in one counted item.

If the column allows NULL, then all rows holding NULL in the column are excluded from the count. If you must count them, it can be done:

**Symbols** [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

```
select count(distinct table.field) +
(select count(*) from rdb$database
where (select count(*) from table t where t.field is null)>0)
from table
```

or in a more standard way:

```
select count(distinct table.field) +
(select count(*) from rdb$database
where exists (select * from table t where t.field is null))
from table
```

- Count(*) can use an index if a WHERE clause is specified and the optimizer can match the WHERE criterion to it, For example, in the statement

```
Select count(*) from employee
where last_name between 'A%' and 'M%';
COUNT is still costly, but less so.
```

The GROUP BY clause

When a query includes a GROUP BY clause, the output of the initial

```
SELECT <input-set> FROM <table-name> [WHERE...]
```

is passed to a further stage of processing, where the GROUP BY clause causes the data to be partitioned into one or more nested groups. Each partition typically extracts summaries by way of expressions that aggregate numeric values at one or more levels. This type of query is known as a *grouped query* and its output is often referred to as a *grouped set*.

Syntax

```
SELECT <input-set> FROM <table-name> [WHERE...]
GROUP BY <grouping-column> [COLLATE collation-order]
HAVING <grouping-column predicate>
[ORDER BY ...];
```



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Input set

The input set that is specified for the SELECT portion of a grouped query has these characteristics:

- it may contain only columns that participate in the grouping defined
- it must match the output of the GROUP BY clause in both degree (number of columns)
- its columns must match the data type and other attributes of their counterparts in the output set

For example, in the following query

```
SELECT PRODUCT_TYPE, SUM(NUMBER SOLD) AS SUM_SALES  
FROM TABLEA  
WHERE SERIAL_NO BETWEEN 'A' AND 'K'  
GROUP BY PRODUCT_TYPE;
```

the output might be similar to the following:

PRODUCT_TYPE	SUM_SALES
-----	-----
Gadgets	174
Whatsits	25
Widgets	117

Grouping column

Notice that, in the preceding query, the GROUP BY clause groups on PRODUCT_TYPE, a column which is present in the input list. It is necessary for grouped queries to select only columns on which any groupings specified can be aggregated. The following query

```
SELECT PRODUCT_ID, PRODUCT_TYPE, SUM(NUMBER SOLD) AS SUM_SALES  
FROM TABLEA  
WHERE SERIAL_NO BETWEEN 'A' AND 'K'  
GROUP BY PRODUCT_TYPE;
```

will fail, because PRODUCT_ID cannot be aggregated within the PRODUCT_TYPE group.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

It works if PRODUCT_ID is included in both the input list and the GROUP BY clause:

```
SELECT PRODUCT_TYPE, PRODUCT_ID, SUM(NUMBER SOLD) AS SUM_SALES  
FROM TABLEA  
WHERE SERIAL_NO BETWEEN 'A' AND 'K'  
GROUP BY PRODUCT_TYPE, PRODUCT_ID;
```

In this case, the aggregation of NUMBER_SOLD moves to the PRODUCT_ID group:

PRODUCT_TYPE	PRODUCT_ID	SUM_SALES
-----	-----	-----
Gadgets	23561	85
Gadgets	45789	21
Gadgets	62345	68
Whatsits	12356	25
Widgets	10089	60
Widgets	41234	57

The HAVING sub-clause

The HAVING clause is a filter for the *grouping output*. This should not be confused with the WHERE clause, since it is applied only to the input set that is generated as output from SELECT....WHERE... You can predicate HAVING criteria just as you do with WHERE criteria.

In the next query, HAVING is used to filter out candidate groups whose PRODUCT_TYPE begins with any letter past 'P':



Symbols [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

```
SELECT PRODUCT_TYPE, PRODUCT_ID, SUM(NUMBER SOLD) AS SUM_SALES  
FROM TABLEA  
WHERE SERIAL_NO BETWEEN 'A' AND 'K'  
GROUP BY PRODUCT_TYPE, PRODUCT_ID  
HAVING PRODUCT_TYPE < 'p', PRODUCT_ID;
```

This eliminates 'Whatsits' and 'Widgets' and produces output just like the first three rows of the previous query's output.

The COLLATE sub-clause

If you want a text grouping column to use a different collation order from the one defined as default for it, you can include a COLLATE clause. For more information about COLLATE, see [The COLLATE clause](#) on page 318 in chapter 17.

The ORDER BY clause

The ORDER BY clause is used for sorting the output of a SELECT query. The clause follows the WHERE clause (if any) and follows the GROUP BY clause (if any). It is available to SELECT queries which output sets to a client or to a cursor.

The general form of the ORDER BY clause is

```
..  
ORDER BY <list of sorting columns>
```

Considerations

- The <list of sorting columns> can be replaced by positive integers representing the degree (left-to-right position) of the output columns. This is useful when the output set involves a JOIN on two or more tables or when sorting on an aliased or calculated column is required.

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

- If the <list of sorting columns> consists only of column names, then it may include columns which are not present in the output set
- if the output set involves a JOIN and column names are used for <list of sorting columns>, then each column in <list of sorting columns> must be qualified by the table alias used in the SELECT list or the table name, if full table qualifiers were used instead of table aliases.
- Ordered sets are costly to server resource and to performance in general. The importance of good indexing to speed up and rationalize the sorting process cannot be too highly emphasized.
- the system administrator should ensure that sufficient temporary storage space is available on the server's disks to accommodate the intermediate output for sorts. For more information refer to entries for the [INTERBASE_TMP](#) and [TMP](#) environment variables and the [tmp_directory](#) parameter in the Firebird configuration file.

Examples

The following two statements are equivalent:

```
SELECT COLA, COLB, COLC, COLD  
FROM TABLEA  
ORDER BY COLA, COLD;
```

and

```
SELECT COLA, COLB, COLC, COLD  
FROM TABLEA  
ORDER BY 1, 4;
```

The next statement produces an ordered set from a JOIN of two tables:

```
SELECT A.COLA, A.COLB, B.COL2, B.COL3  
FROM TABLEA A  
JOIN B TABLEB ON A.COLA = B.COL1  
WHERE A.COLX = 'Sold'  
ORDER BY A.COLA, B.COL3;
```

For more information about ordering sets, refer to the *Firebird Reference Guide*—[SQL Statement and Function Reference](#) (ch. 2 p. 43).



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

The UNION operator

UNION is used to combine the output sets of two or more SELECTs in a single statement to produce a single, read-only set which may contain rows derived from different tables. The query must be constructed in such a way as to make each contributing input set *union compatible*.

Union compatible sets

For each SELECT operation contributing an output list as input to a UNION set, the output set must consist of a column list which exactly matches those of all of the others by column identifier, degree (number and order of columns) and respective data type. For example, assuming TABLEA and TABLEB both contain corresponding columns COL_ID, COLA, COLB and COLC having identical specifications, then the tables are union compatible:

```
SELECT COL_ID, COLA, COLB, COLC FROM TABLEA  
UNION  
SELECT COL_ID, COLA, COLB, COLC FROM TABLEB;
```

For this reason, `SELECT * FROM table_name` in any contributing set will not suffice if column names do not match in all tables.

If it is possible to match data types but not column identifiers, column aliasing can be used in the output lists of all contributing SELECT clauses. For example,

```
SELECT COL_ID AS ID_COL, COLA AS ACOL, COLB AS BCOL, COLC AS CCOL  
FROM TABLEA  
UNION  
SELECT ID_COL, COL1 AS COLA, COL2 AS BCOL, COL3 AS CCOL  
FROM TABLEB;
```

UNION ALL

If duplicate rows are found during the creating of the union set, the default behavior is to exclude the duplicate rows from the set. To include the duplicates, use UNION ALL instead of UNION on its own.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

The DISTINCT keyword

In circumstances where the structure of the specified output set might produce duplicate row where you need an output set containing no identical rows—as may occur, for example, when the specified output set the primary key or joins tables in one-to-many relationships—use the DISTINCT keyword with SELECT to eliminate the duplicates. For example,

```
SELECT LAST_NAME FROM EMPLOYEE  
WHERE LAST_NAME = 'Smith';
```

will produce exactly one row, regardless of how many employee records have LAST_NAME = 'Smith'.

Considerations

- SELECT DISTINCT is an aggregate operation which cannot return any “updatable” rows. That is to say, no single row in the output set can uniquely identify the row from which it derived.
- For the same reason, a view that is specified with SELECT DISTINCT is read-only. For more information on this topic, see [Read-only and updatable views](#) on p. 368 of chapter 19.

Column aliasing

Firebird supports the SQL standard which

- allows any column to be output using an alias of one or more characters
- requires columns which are computed or sub-selected in the statement to be explicitly named with an alias

For example,

```
SELECT COL_ID, COLA||','||COLB AS comput_col  
FROM TABLEA;
```

Firebird permits the standard to be relaxed slightly in two respects:

- the AS keyword is optional.
 This is harmless, but not particularly to be recommended, since it can make aliased column names harder to find in source code
- aliasing of computed or sub-selected columns may be omitted.

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

For example, the following query:

```
SELECT cast(CURRENT_DATE as VARCHAR(10))||'-'|| registration  
FROM AIRCRAFT;
```

generates the following output:

```
<blank title line>  
=====  
2002-04-30-GORILLA
```

- ❖ This is certainly not recommended—it causes a column to be output with no column name. Most drivers and connectivity components treat this as a bug and it is regarded by many developers as such.

Internal functions

 For full descriptions, see *Firebird Reference Guide*—[SQL Statement and Function Reference](#) (ch. 2 p. 43).

The following SQL internal functions are available in Firebird. .

TABLE 9–2 SQL functions

Function	Type	Purpose
AVG()	Aggregate	Calculates the average of a set of values
CAST()	Conversion	Converts a column from one datatype to another
COUNT()	Aggregate	Returns the number of rows that satisfy a query's search condition
EXTRACT()	Conversion	Extracts data and time information from DATE, TIME, and TIMESTAMP values
FIRST(m) SKIP(n)	Aggregate	Retrieves a subset of rows from a SELECTED set



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

TABLE 9–2 SQL functions (*continued*)

Function	Type	Purpose
<u>GEN_ID()</u>	Numeric	Returns a system-generated value
<u>MAX()</u>	Aggregate	Retrieves the maximum value from a set of values
<u>MIN()</u>	Aggregate	Retrieves the minimum value from a set of values
<u>SUBSTRING()</u>	String	Retrieves any sequence of characters from a string
<u>SUM()</u>	Aggregate	Totals the values in a set of numeric values
<u>UPPER()</u>	Conversion	Converts a string to all uppercase

Aggregate functions perform calculations over a series of values, such as the columns retrieved with a SELECT statement.

Conversion functions transform datatypes, either converting them from one type to another, or by changing the scale or precision of numeric values, or by converting CHARACTER datatypes to all uppercase.

The **numeric function**, GEN_ID(), produces a system-generated number that can be inserted into a column requiring an integer datatype.

SQL operators and expressions

Firebird SQL syntax includes operators and expressions for comparing and evaluating columns, constants, and host-language variables to produce a single value. SQL expressions can be used in the following contexts:

- definition of computed output columns
- WHERE clause search conditions
- DELETE and UPDATE search conditions
- INSERT VALUE clause
- UPDATE SET clause
- GROUP BY clause (user-defined functions only)



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Using expressions to define columns

Any expression that performs a comparison or computation and returns a single value can be involved in the specification for a column in an output set.

Examples of expressions are concatenating character strings, performing computations on numeric data, doing comparisons using comparison operators (<, >, <=, and so on) or Boolean operators (AND, OR, NOT).

The expression

- must return a single value
- cannot be an array or return an array
- must not refer to any column which does not exist in the table

The following table describes the elements that can be used in SQL expressions.

TABLE 9-3 Elements of SQL expressions

Element	Description
Column names	Columns from specified tables, against which to search or compare values, or from which to calculate values.
Host-language variables	Program variables containing changeable values.
Constants	Hard-coded numbers or quoted strings, such as 507 or 'Tokyo'.
Concatenation operator	, used to combine character strings.
Arithmetic operators	+, -, *, and /, used to calculate and evaluate values.
Logical operators	Reserved words, NOT, AND, and OR, used within simple search conditions, or to combine simple search conditions to make complex searches. A logical operation evaluates to true or false. Usually used only in search conditions.
Internal context variables	"Pseudo-column" variables maintained by the server



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

TABLE 9–3 Elements of SQL expressions (*continued*)

Element	Description
CAST(value AS type)	Expressions explicitly cast a value of one data type to another data type
COLLATE clause	Comparisons of CHAR and VARCHAR values can sometimes take advantage of a COLLATE clause to force the way text values are compared.
Date literals	String-like values that can be entered in quotes, that will be interpreted as date values in EXTRACT, SELECT, INSERT, and UPDATE operations. Possible strings are 'TODAY', 'NOW', 'YESTERDAY', and 'TOMORROW'. ❖ In Dialect 3, these literals must be CAST as a valid date/time type when used in EXTRACT and SELECT expressions.
Comparison operators	<, >, <=, >=, =, and <>, used to compare a value on the left side of the operator to another on the right. A comparative operation evaluates to true or false. Other, more specialized comparison operators include ALL, ANY, BETWEEN, CONTAINING, EXISTS, IN, IS [NOT] NULL, LIKE, SINGULAR, SOME, and STARTING WITH. These operators can evaluate to true, false, or unknown. They are usually used only in search conditions.
Stored procedures	Reusable SQL statement blocks that can receive and return parameters, and that are stored as part of a database's metadata.

**Symbols** [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)TABLE 9–3 Elements of SQL expressions (*continued*)

Element	Description
Subqueries	SELECT statements, typically nested in WHERE clauses, that return values to be compared with the result set of the main SELECT statement.
Parentheses	Used to group expressions into hierarchies; operations inside parentheses are performed before operations outside them. When parentheses are nested, the contents of the innermost set is evaluated first and evaluation proceeds outward.

Server context variables

Firebird supplies a range of variables that will supply server snapshot values which can be stored into database columns.

TABLE 9–4 Firebird server context variables

Context Variable	Data Type	Description
USER	VARCHAR(128)	References the name of the user who is currently logged in. For example, USER can be used as a default in a column definition or to enter the current user's name in an INSERT. When a user name is present in a table, it can be referenced with USER in SELECT and DELETE statements.
CURRENT_USER	VARCHAR(128)	Same as USER.
CURRENT_ROLE	VARCHAR(128)	Role under which the CURRENT_USER is logged in. If the role does not exist, it is reset to NONE without returning an error.
CURRENT_TIMESTAMP	TIMESTAMP	Registers the current server time.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Precedence of operators

Precedence determines the order in which operators and the values they affect are evaluated in an expression. Table 9–5 lists the precedence of Firebird operator types, from highest to lowest.

TABLE 9–5 Operator type precedence

Operator type	Precedence	Explanation
Concatenation	1	Strings are concatenated before all other operations take place.
Arithmetic	2	Arithmetic is performed after string concatenation, but before comparison and logical operations.
Comparison	3	Comparison operations are evaluated after string concatenation and math, but before logical operations.
Logical	4	Logical operations are evaluated after all other operations.

When an expression contains several operators of the same type, those operators are evaluated from left to right unless there is a conflict where two operators of the same type affect the same values. When there is a conflict, operator precedence within type determines the order of evaluation.

Concatenation operator

The concatenation operator || combines two character strings to produce a single string. Character strings can be constants or values retrieved from a column:

```
SELECT Last_name || ', ' || First_Name AS FullName FROM Student;
```



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

Arithmetic operators

Firebird supports the four arithmetic operators listed in the following table:

TABLE 9–6 Arithmetic operator precedence

Operator	Purpose	Precedence
*	Multiplication	1
/	Division	2
+	Addition	3
-	Subtraction	4

Arithmetic expressions are evaluated from left to right, except when ambiguities arise. In these cases, arithmetic operations are evaluated according to the precedence specified in Table 9–6 (for example, multiplications are performed before divisions, and divisions are performed before subtractions).

Arithmetic operations are always performed before comparison and logical operations. To change or force the order of evaluation, group operations in parentheses.

Comparison operators

Comparison operators test a specific relationship between a value on the left of the operator, and a value or range of values to the right of the operator. Every test evaluates to a Boolean TRUE or FALSE. Values compared must evaluate to the same data type. The CAST() function can be used to translate a value to a compatible data type for comparison. Values can be columns, constants, or calculated expressions.

- ★ Firebird comparison operators support comparisons between a value on the left of the operator with the results of a subquery on the right of the operator.

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

The following table shows the evaluation order for comparison operators, from highest to lowest.

TABLE 9–7 Comparison operator precedence

Operator	Purpose	Precedence
=, ==	Equal to, identical to	1
<>, !=, ~=, ^=	Not equal	2
>	Greater than	3
<	Less than	4
>=	Greater than or equal to	5
<=	Less than or equal to	6
!>, ~>, ^>	Not greater than	7
!<, ~<, ^<	Not less than	8
ALL	Equals all values returned by a subquery	9
ANY	Equals any value returned by a subquery	9
BETWEEN	Value falls within a range of values	9
CONTAINING	String value contains specified string	9
EXISTS	Exists in <i>at least one</i> value returned by a subquery	9
IN	Exists in <i>at least one</i> value in a list	9
LIKE	Equals specified string, with wildcard substitution	9
NULL	Absence of a value	9
SINGULAR	Exists in <i>exactly one</i> value returned by a subquery	9



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

TABLE 9–7 Comparison operator precedence (*continued*)

Operator	Purpose	Precedence
SOME	Equals any values returned by a subquery	9
STARTING WITH	String value begins with specified string	9

❖ Comparisons evaluate to NULL if a NULL value is encountered on either the right or the left of the operator.

During normal left-to-right evaluation ALL, ANY, BETWEEN, CONTAINING, EXISTS, IN, LIKE, NULL, SINGULAR, SOME, and STARTING WITH are evaluated after the other comparison operators listed above. When they conflict with one another they are evaluated strictly from left to right.

Logical operators

Firebird recognizes the three logical operators listed in the table below::

TABLE 9–8 Logical operator precedence

Operator	Purpose	Precedence
NOT	Negate a condition	1
AND	Combine two conditions: both must be true	2
OR	Combine two conditions: at least one must be true	3

NOT negates the search condition to which it is applied.

AND and OR combine two or more search conditions.



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

For a logical predicate to be true

- all of the **AND**-ed search conditions must be true
- at least one **OR**-ed condition must be true. **OR** operations are evaluated left-to-right with short-circuiting; that is, if the first term evaluates true, the second term is not evaluated.

Inclusive vs exclusive OR

OR is an **inclusive OR** (any single condition or all conditions can be true).

Firebird does not support an **exclusive OR** (evaluates to true if any single condition is true but false if all conditions are true).

Precedence of logical operations

When search conditions are combined, the order of evaluation is determined by precedence of the operators that connect them. A **NOT** condition is evaluated before **AND**, and **AND** is evaluated before **OR**. Parentheses can be used to change the order of evaluation.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

CHAPTER 10

Interactive SQL Utility (isql)

The isql command-line SQL utility, installed in the /bin directory of your Firebird installation, provides a non-graphical interface for interactive access to a Firebird or compatible InterBase(R) database that is consistent on all server and client platforms.

It can be used for a number of administrative tasks, including the creation and running of scripts for creating and modifying metadata.



OTHER REFERENCES

For a complete alphabetical summary of Firebird isql commands, see *Firebird Reference Guide*—[isql command reference](#) (ch. 1 p. 1).

For a description of the standard SQL commands available in isql, see [Statement list](#) on page 115 in chapter 9 and *Firebird Reference Guide*—[SQL Statement and Function Reference](#) (ch. 2 p. 43).

Modes for isql Use

You can invoke **isql** in a range of modes.-

- as an *interactive session* from the command-line of your operating system shell. An interactive session can be invoked in a number of ways and lasts until you terminate the session using a QUIT or EXIT command

Use **isql** interactively to

- create, update, query and drop data and metadata
- input a file ("script") containing a batch of SQL statements for execution in sequence without prompting

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

- add and modify data
 - grant user permissions
 - test queries
 - perform some database administration tasks
-
- directly from the command-line, with individual options, without starting an interactive session. Commands execute and, upon completion, return control automatically to your operating system shell.
 - from a shell script or batch file, with individual options

Invoking isql

To start the isql utility, type the following at a Linux/UNIX shell prompt or Windows console prompt:

```
isql [options] [database_name]
```

where *options* are command-line options and *database_name* is the name of the database to connect to, including disk and directory path.

❖ Unless you have the ISC_USER and ISC_PASSWORD declared as operating system variables, your options when invoking isql will need to include an appropriate -user and -password, e.g.

```
isql -user TEMPDBA -password osoweary
```

- If no options are specified, **isql** starts an interactive session.
- If no database is specified, you must connect to an existing database or create a new one after starting isql.
- If a database was specified, **isql** starts the interactive session by connecting to the named database, provided the login options are correct and valid for that database.

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

- The options specified will determine whether **isql** starts interactively or noninteractively.

For example, reading an input file and writing to an output file are not interactive tasks, so the **-input** or **-output** command-line options do not start an interactive session. The options **-a**, **-database**, **-extract** and **-x**, which are used when extracting DDL statements, are also non-interactive.

The terminator character

The default terminator is the semicolon (:), which is used for all of the examples in this chapter. You can change the terminator to any character or group of characters with the **SET TERMINATOR** command or with the **-terminator** command-line option.

Connecting to a database

If you do not specify a database on the command-line when invoking **isql**, you must either connect to an existing database or create a new one. Use the **CONNECT** command to connect to a database and **CREATE DATABASE** to create a database.

You can connect to a database in two ways:

- **locally**, to a database on Windows NT/2000 or Windows 95/98.

Use the **CONNECT** command with the full path of the database as the argument. For example:

```
SQL> CONNECT 'C:/Firebird/examples/employee.gdb' role 'magician';
```

- **from a remote workstation**, to a database on a Windows or Linux/UNIX server using TCP/IP. Use the **CONNECT** command with the full server host name and path of the database as the argument. Separate the server host name from the database path with a colon.

To connect to a database on a Linux/UNIX server named phoenix:

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

```
SQL> CONNECT 'phoenix:/usr/firebird/examples/employee.gdb';
```

- To connect to a database on a Windows NT or Win2K server named phoenix:

```
SQL> CONNECT 'phoenix:c:/Firebird/examples/database/employee.gdb';
```

For the full syntax of CONNECT and CREATE DATABASE, see the **Statement List** in the *Language Elements* chapter of this volume.

❖ On Windows, don't confuse server names and shared disks. Both are specified with a colon separator. The client/server network layer does not recognize mapped drives. If you specify a single letter that maps to a disk drive, it is assumed to be a drive, not a server name.

★ You can use either forward slashes (/) or backslashes (\) as directory separators. Firebird automatically converts either type of slash to the type appropriate for the server's operating system.

Transaction behavior in isql

When you start **isql**, Firebird begins a transaction. That transaction remains in effect until you issue a COMMIT or ROLLBACK statement. You must issue a COMMIT or ROLLBACK statement to end a transaction.

Issuing one of these statements automatically starts a new transaction. You can also start a transaction with the SET TRANSACTION statement.

isql uses a separate transaction for interactive DDL statements. When these statements are issued at the **SQL>** prompt, they are committed automatically as soon as they are completed.

For more information , see [Committing Statements in Scripts](#) on p. 230 of chapter 12.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Using warnings

Warnings can be issued for the following conditions:

- SQL statements with no effect
- SQL expressions that produce different results in InterBase 5 versus Firebird 1
- API calls that will be replaced in future versions of the product
- Pending database shutdown

Warnings are toggled on by default when an **isql** command or session is initiated. To suppress warnings, use the `-nowarnings` command-line option. For toggling the display off and on during an interactive **isql** session, refer to the **isql** command `SET WARNINGS (SET WNG)`.

Error handling

Firebird handles errors in **isql** and DSQL in the same way. To indicate the causes of an error, **isql** displays an error message consisting of the `SQLCODE` variable and the Firebird status array.

TABLE 10-1 **isql** error codes and messages

SQLCODE	Message	Meaning
<0	SQLERROR	Error occurred: statement did not execute
0	SUCCESS	Successful execution
+1—99	SQLWARNING	System warning or information message
+100	NOT FOUND	No qualifying rows found, or end of current active set of rows reached



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Setting dialect in isql

If you start **isql** and attach to a database without specifying a dialect, **isql** takes on the dialect of the database.

You can set the **isql** dialect in the following ways:

- From the command line when invoking **isql**

```
isql -sql_dialect n ;
```

where *n* is 1, 2, or 3.

If you specify the dialect this way, **isql** retains that dialect after connection unless you explicitly change it.

- Within an **isql** session or in a SQL script

```
SET SQL DIALECT n ;
```

★ When you change the dialect during a session using **SET SQL DIALECT n**, from the **SQL>** prompt or by running and committing a script containing that statement, **isql** continues to operate in that dialect unless it is explicitly changed.

- When you create a database using **isql**:

- If **isql** was set to dialect 1 and you created a database interactively you would get a dialect 1 database.
- If you created a database interactively without first specifying a dialect for the **isql** session and without attaching to a database, or by using a DDL script in these conditions that did not begin with a **SET SQL DIALECT** statement, **isql** creates the database in dialect 3.

❖ Important

- Any Firebird 1 **isql** client that attaches to an InterBase 5 database resets to dialect 1.
- You cannot set dialect in a **CREATE DATABASE** statement



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

TABLE 10-2 Precedence of dialect settings in isql

Precedence	Description
Lowest	Dialect of an attached Firebird 1 database
Next lowest	Dialect specified on the command line
Next highest	Dialect specified during the session
Highest	Dialect of an attached InterBase 5 database (= 1)

Interactive commands in isql

You can enter three kinds of commands or statements interactively at the `SQL>` prompt:

- **SQL data definition** (DDL) statements, such as CREATE, ALTER, DROP, GRANT and REVOKE. These statements create, modify or remove metadata and objects or control user access permission (privileges) to the database.
- **SQL data manipulation** (DML) statements such as SELECT, INSERT, UPDATE and DELETE. These DML operations affect the data in a database. The output of SELECT commands can be displayed or directed to a file.
- **isql** commands, which fall into three main categories:
 - General commands (for example, commands to read an input file, write to an output file, or end an `isql` session)
 - SHOW commands (to display metadata or other database information)
 - SET commands (to modify the `isql` environment)



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Examples of starting an interactive isql session

This starts an interactive connection to a remote database. The remote server, phoenix, accepts the specified user and password combination with the privileges assigned to the MAGICIAN role:

```
isql -user monkey -password banana -role 'magician'  
'phoenix:/usr/miracles.gdb'
```

This starts an interactive session but does not attach to a database. **isql** commands are displayed, and query results print column headers every 30 lines:

```
isql -echo -page 30
```

When you start an interactive **isql** session, the following *interactive prompt* appears:

```
SQL>
```

Once the interactive prompt appears, **isql** expects each command to end with a terminator character. Until you type in the terminator, **isql** will display a continuation prompt each time you press the ENTER key to input what you have typed:

```
CON>
```

Creating and altering database objects

In an **isql** session, or through the command line, you can submit DDL statements one-by-one to CREATE or DROP databases, domains, generators, tables, indexes, triggers and stored procedures. With the exception of generators, you can also issue ALTER statements for any of these objects.

While it is possible to build a database by submitting and committing a series of DDL statements during an interactive **isql** session, this approach is *ad hoc*. It leaves you with no documentation of the process and potential holes in your QA review process.

It is very good practice to use a *script* to create your database and its objects. A script for creating and altering database objects is sometimes known as a data definition file or, more commonly, a *DDL script*.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- ★ DDL is a mnemonic for *Data Definition Language*, the subset of SQL used for defining and altering database objects.

Experienced Firebird programmers often create a set of DDL scripts, designed to run and commit in a specific order, to make debugging easy and ensure that objects will exist when later, dependent objects refer to them.

Scripting ongoing changes in a DDL script also gives you a permanent record of the evolution of your database and the ability to review and reverse any imprudent changes.

- ★ A script is just a plain text file created in the correct format for the filesystem on which it is to be run. You can write a script yourself using an editor such as *vi* or *ded* (on Linux) or *Edit.exe* or *Notepad.exe* (on Windows). **Isql** can create a script during an interactive session by passing your keyboard input to a named file. Read notes on the OUTPUT command for more information.

Use the INPUT command to run DDL scripts in an interactive **isql** session or from the command line.

- ★ A script does not have to have been created in **isql** in order to run in **isql**.

- BOOK For more details about DDL scripts, see the next chapter [Writing and Running Scripts](#).

General **isql** commands

The general **isql** commands perform a variety of useful tasks, including reading a SQL script, executing shell commands, and exiting **isql**.

These commands are: BLOBDUMP, EDIT, EXIT, HELP, INPUT, OUTPUT, QUIT AND SHELL.. Descriptions appear below.



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

BLOBDUMP

stores BLOB data into a named file

Syntax

```
SQL> BLOBDUMP blob_id filename ;
```

TABLE 10-3 BLOBDUMP arguments

Argument	Description
blob_id	Identifier consisting of two hex numbers separated by a colon (:). The first number is the ID of the table containing the BLOB column, the second is a sequenced instance number. To get the blob_id, issue any SELECT statement that selects a column of BLOB data. The output will show the hex blob_id above or in place of the BLOB column data, depending on whether SET BLOB[DISPLAY] is ON or OFF.
filename	Fully qualified filesystem name of the file which is to receive the data.

Example SQL> BLOBDUMP 32:d48 IMAGE.JPG ;

See Also *Firebird Reference Guide*—[BLOBDUMP](#) (ch. 1 p. 1)

EDIT

allows editing and re-execution of the previous isql command or of a batch of commands in a source file.

Syntax



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

SQL> EDIT [filename] ;

TABLE 10-4 EDIT arguments

Argument	Description
filename	Optional, fully qualified filesystem name of file to edit.

Example SQL> EDIT /usr/mystuff/batch.sql

or, to open the previous statement in the editor:

```
SQL> SELECT EMP_CODE, EMP_NAME FROM EMPLOYEE ;
SQL> EDIT ;
```

See Also *Firebird Reference Guide*—[EDIT](#) (ch. 1 p. 2)

EXIT

commits the current transaction without prompting, closes the database and ends the **isql** session.

❖ If you need to roll back the transaction instead of committing it, use **QUIT** instead.

Syntax

```
SQL> EXIT ;
```

No arguments.

HELP

displays a list of **isql** commands with descriptions. You can combine it with **OUTPUT** to print the list to a file.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Syntax

```
SQL> HELP ;
```

Example

```
SQL> OUTPUT HELPLIST.TXT ;
SQL> HELP ;
SQL> OUTPUT ; /* toggles output back to the monitor */
```

No arguments.

INPUT

reads and executes a block of commands from a named text file (SQL script). Input files can embed other INPUT commands, thus providing the capability to "chain" DDL scripts. To create scripts, use a text editor or, to build them interactively, use the OUTPUT command.

Syntax

```
SQL> INPUT filename ;
```

TABLE 10-5 INPUT arguments

Argument	Description
filename	Name of a file containing SQL statements and commands.

Example SQL> INPUT myscript.ddl ;

See Also *Firebird Reference Guide*—[INPUT](#) (ch. 1 p. 5)



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

OUTPUT

redirects output to a disk file or (back) to the standard output device (monitor). To output both data and commands, use SET ECHO ON. To output data only, use SET ECHO OFF.

Syntax

```
SQL> OUTPUT [filename] ;
```

TABLE 10-6 OUTPUT arguments

Argument	Description
filename	Name of the file in which to save output. If no file name is given, results appear on the standard monitor output.

Example

```
SQL> OUTPUT employees.dta ;
SQL> SELECT EMP_NO, EMP_NAME FROM EMPLOYEE ; /* output goes to file */
SQL> OUTPUT ; /* toggles output back to the monitor */
```

See Also *Firebird Reference Guide*—[OUTPUT](#) (ch. 1 p. 7)

QUIT

rolls back the current transaction without prompting, closes the database and ends the **isql** session.

❖ If you need to commit the transaction instead of rolling it back, use EXIT instead.

Syntax

```
SQL> QUIT ;
```

No arguments.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

SHELL

(from an interactive session) gives temporary access to the command line of the operating system shell without committing or rolling back any transaction.

Syntax

```
SQL> SHELL [operating system command] ;
```

TABLE 10-7 SHELL arguments

Argument	Description
operating system command	A valid operating system command or call. The command will be executed and control returns to isql . If no command is specified, isql opens an interactive session in the OS shell. Typing exit returns control to isql .

Example SQL> SHELL dir /mydir ;

See Also Firebird Reference Guide—[SHELL](#) (ch. 1 p. 25)

SHOW commands

SHOW commands are (approximately) the interactive equivalent of the command-line -extract, -x or -a options. They are used to display metadata, including tables, indexes, procedures, and triggers. They list all of the specified objects or give information about a particular object named in the command.

- ❖ Although you can use the OUTPUT command to send the output of the SHOW commands to a file, the resulting text will not be suitable for use as a DDL script without editing. Use the command-line option if obtaining DDL scripts is your goal.

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

SHOW commands operate on a separate transaction from user statements. They run as READ COMMITTED background statements and acknowledge all metadata changes immediately.

SHOW CHECK

displays all user-defined CHECK constraints defined for a table.

Syntax

```
SQL> SHOW CHECK tablename ;
```

TABLE 10-8 SHOW CHECK arguments

Argument	Description
tablename	Name of a table that exists in the attached database

Example

```
...
SQL> SHOW CHECK JOB ;  
  
CHECK (min_salary < max_salary)
```

SHOW DATABASE

displays information about the attached database (file name, page size and allocation, sweep interval, transaction numbers and Forced Writes status). For version and on-disk structure information, see SHOW VERSION.

Syntax

```
SQL> SHOW DATABASE | DB ;
```



Symbols **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

No arguments.

Shorthand: SHOW DB is equivalent to SHOW DATABASE.

Example

...

```
SQL> SHOW DB ;
```

```
Database: employee.gdb
          Owner: SYSDBA
PAGE_SIZE 4096
Number of DB pages allocated = 422
Sweep interval = 20000
```

SHOW DOMAIN[S]

displays domain information.

Syntax

```
SQL> SHOW { DOMAINS | DOMAIN name } ;
```

TABLE 10–9 SHOW DOMAIN(S) arguments

Argument	Description
DOMAINS	Lists the names of all the domains declared in the database
DOMAIN <i>name</i>	Displays definition of the named single domain

Examples



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

...

```
SQL> SHOW DOMAINS ;
```

D_CURRENCY	D_NOTES
D_BOOLEAN	D_PHONEFAX
...	...

```
SQL> SHOW DOMAIN D_BOOLEAN ;
```

D_BOOLEAN	SMALLINT NOT NULL
	DEFAULT 0
	CHECK (VALUE IN (0,1))

SHOW EXCEPTION[S]

displays exception information.

Syntax

```
SQL> SHOW { EXCEPTIONS | EXCEPTION name } ;
```

TABLE 10-10

Argument	Description
EXCEPTIONS	Lists the names and texts of all exceptions declared in the database
EXCEPTION <i>name</i>	Displays text of the named single exception

Examples



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

...

```
SQL> SHOW EXCEPTIONS ;
```

```
Exception Name    Used by,  Type
=====
BAD_WIZ_TYPE     UPD_FAVEFOOD, Stored procedure
                  Invalid Wiz type, check CAPS LOCK
...
```

```
SQL> SHOW EXCEPTION BAD_WIZ_TYPE ;
```

```
Exception Name    Used by,  Type
=====
BAD_WIZ_TYPE     UPD_FAVEFOOD, Stored procedure
                  Invalid Wiz type, check CAPS LOCK
```

SHOW FUNCTION[S]

displays information about user-defined functions (UDFs) declared in the attached database.

Syntax

```
SQL> SHOW { FUNCTIONS | FUNCTION name } ;
```

TABLE 10-11 SHOW FUNCTION(S) arguments

Argument	Description
FUNCTIONS	Lists the names of all UDFs declared in the database
FUNCTION <i>name</i>	Displays the declaration of the named UDF



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

Examples

...

```
SQL> SHOW FUNCTIONS ;
```

ABS	MAXNUM
LOWER	SUBSTRLEN
...	...

```
SQL> SHOW FUNCTION maxnum ;
```

```
Function MAXNUM:  
Function library is /usr/firebird/udf/ib_udf.so  
Entry point is FN_MAX  
Returns BY VALUE DOUBLE PRECISION  
Argument 1: DOUBLE PRECISION  
Argument 2: DOUBLE PRECISION
```

SHOW GENERATOR[S]

displays information about generators declared in the attached database.

Syntax

```
SQL> SHOW { GENERATORS | GENERATOR name } ;
```

TABLE 10-12 SHOW GENERATOR(S) arguments

Argument	Description
GENERATORS	Lists the names of all generators declared in the database, along with their next values
GENERATOR <i>name</i>	Displays the declaration of the named generator, along with its next value



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)
[Examples](#)

...

```
SQL> SHOW GENERATORS ;
```

```
Generator GEN_EMPNO, Next value: 1234  
Generator GEN_JOBNO, Next value: 56789  
Generator GEN_ORDNO, Next value: 98765
```

... ...

```
SQL> SHOW GENERATOR gen_ordno ;
```

```
Generator GEN_ORDNO, Next value: 98765
```

SHOW GRANT

displays privileges and ROLE ownership information about a named object in the attached database; or displays user membership within roles.

Syntax

```
SQL> SHOW GRANT { object | rolename } ;
```

TABLE 10-13 SHOW GRANT arguments

Argument	Description
object	Name of an existing table, view or procedure in the current database
rolename	Name of an existing role in the current database. Use SHOW ROLES to list all the roles defined for this database.



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)
Examples

...

```
SQL> SHOW GRANT JOB ;  
GRANT SELECT ON JOB TO ALL  
GRANT DELETE, INSERT, SELECT, UPDATE ON JOB TO MANAGER  
  
SQL> SHOW GRANT DO_THIS ;  
GRANT DO_THIS TO MAGICIAN
```

SHOW INDEX (SHOW INDICES)

displays information about a named index, about indices for a specified table or about indices for all tables in the attached database.

Syntax

```
SQL> SHOW { INDICES | INDEX { index | table } } ;
```

TABLE 10-14 SHOW INDEX arguments

Argument	Description
<i>index</i>	Name of an existing index in the current database
<i>table</i>	Name of an existing table in the current database

Shortcut: SHOW IND is an alias for either SHOW INDEX or SHOW INDICES

Examples



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

...

```
SQL> SHOW INDEX ;  
RDB$PRIMARY1 UNIQUE INDEX ON COUNTRY(COUNTRY)  
CUSTNAMEX INDEX ON CUSTOMER(CUSTOMER)  
CUSTREGION INDEX ON CUSTOMER(COUNTRY, CITY)  
RDB$FOREIGN23 INDEX ON CUSTOMER(COUNTRY)
```

...

```
SQL> SHOW IND COUNTRY ;  
RDB$PRIMARY20 UNIQUE INDEX ON CUSTOMER(CUSTNO)  
CUSTNAMEX INDEX ON CUSTOMER(CUSTOMER)
```

SHOW PROCEDURE[S]

lists all procedures in the attached database, with their dependencies; or displays the text of the named procedure with the declarations and types (input/output) of any parameters.

Syntax

```
SQL> SHOW { PROCEDURES | PROCEDURE name } ;
```

TABLE 10-15 SHOW PROCEDURE(S) arguments

Argument	Description
name	Name of an existing stored procedure in the current database

Shorthand: SHOW PROC is an alias for either SHOW PROCEDURE or SHOW PROCEDURES

Examples



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

...

SQL> SHOW PROCEDURES ;

Procedure Name	Dependency	Type
ADD_EMP_PROJ	EMPLOYEE_PROJECT	Table
	UNKNOWN_EMP_ID	Exception
DELETE_EMPLOYEE	DEPARTMENT	Table
	EMPLOYEE	Table
	EMPLOYEE_PROJECT	Table

...

SQL> SHOW PROC ADD_EMP_PROJ ;

Procedure text:

```
=====
BEGIN
  BEGIN
    INSERT INTO EMPLOYEE_PROJECT (
      EMP_NO, PROJ_ID)
    VALUES (
      :emp_no, :proj_id) ;
    WHEN SQLCODE -530 DO
      EXCEPTION UNKNOWN_EMP_ID;
    END
    RETURN ;
  END
=====
```

Parameters:

EMP_NO INPUT SMALLINT
PROJ_ID INPUT CHAR(5)



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

SHOW ROLE[S]

displays the names of SQL roles for the attached database.

★ To show user membership within roles, use SHOW GRANT *rolename*.

Syntax

```
SQL> SHOW ROLES ;
```

No arguments.

Examples

...

```
SQL> SHOW ROLES ;
```

MAGICIAN	MANAGER
PARIAH	SLEEPER

...

SHOW SQL DIALECT

displays the SQL dialects of the client and of the attached database, if there is one.

Syntax

```
SQL> SHOW SQL DIALECT;
```

Example

...

```
SQL> SHOW SQL DIALECT;
```

```
Client SQL dialect is set: 3 and database SQL dialect is: 3
```



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

SHOW SYSTEM

displays the names of system tables and system views for the attached database.

Syntax

```
SQL> SHOW SYS [ TABLES ] ;
```

No arguments. TABLES is an optional keyword that does not affect the behavior of the command.

Shorthand: SHOW SYS is equivalent.

Examples

...

```
SQL> SHOW SYS ;
```

RDB\$CHARACTER_SETS	RDB\$CHECK_CONSTRAINTS
RDB\$COLLATIONS	RDB\$DATABASE

...

For more detailed information about the system tables, see the *Firebird Reference Manual*.

SHOW TABLE[S]

lists all tables or views, or displays information about the named table or view.

See also SHOW VIEWS

Syntax



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

SQL> SHOW { TABLES | TABLE *name* } ;

TABLE 10-16 SHOW TABLE(S) arguments

Argument	Description
name	Name of an existing table or view in the current database. <ul style="list-style-type: none">• If the object is a table, the output contains column names and definitions, PRIMARY KEY, FOREIGN KEY and CHECK constraints, and triggers.• If the object is a view, the output contains column names and the SELECT statement that the view is based on.

Examples

```
...
SQL> SHOW TABLES ;
COUNTRY          CUSTOMER
DEPARTMENT       EMPLOYEE
EMPLOYEE_PROJECT JOB
...
SQL> SHOW TABLE COUNTRY ;
COUNTRY  COUNTRYNAME VARCHAR(15) NOT NULL
        CURRENCY VARCHAR(10) NOT NULL
        PRIMARY KEY (COUNTRY)
```

SHOW TRIGGER[S]

displays all triggers defined in the database, along with the table they depend on; or, for the named trigger, displays its sequence, type, activation status (ACTIVE/INACTIVE) and procedure definition.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Syntax

```
SQL> SHOW {TRIGGERS | TRIGGER name} ;
```

TABLE 10-17 SHOW TRIGGER(S) arguments

Argument	Description
name	Name of an existing trigger in the current database

Shorthand: SHOW TRIG is an alias for either SHOW TRIGGER or SHOW TRIGGERS

Examples

```
SQL> SHOW TRIGGERS ;  
Table name          Trigger name  
===== =====  
EMPLOYEE           SET_EMP_NO  
EMPLOYEE           SAVE_SALARY_CHANGE  
CUSTOMER           SET_CUST_NO  
SALES              POST_NEW_ORDER
```



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

...

```
SQL> SHOW TRIG SET_CUST_NO ;
```

Trigger:

```
SET_CUST_NO, Sequence: 0, Type: BEFORE INSERT, Active
AS
BEGIN
    new.custno = gen_id(cust_no_gen, 1);
END
```

SHOW VERSION

displays information about the software versions of isql and the Firebird server program, and the on-disk structure of the attached database.

Syntax

```
SQL> SHOW VERSION ;
```

No arguments.

Shorthand: SHOW VER is equivalent.

Example

...

```
SQL> SHOW VER ;
```

```
ISQL Version: FB-WI-1.0.0.555-RC1
Firebird/Windows NT (access method), version 'FB-WI-6.2.555-RC1'
on disk structure version 10.0
```



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

SHOW VIEW[S]

lists all views, or displays information about the named view.

★ See also SHOW TABLES

Syntax

```
SQL> SHOW { VIEWS | VIEW name } ;
```

TABLE 10-18 SHOW VIEW(S) arguments

Argument	Description
name	Name of an existing view in the current database. The output contains column names and the SELECT statement that the view is based on.

Example

```
SQL> SHOW VIEWS ;
PHONE_LIST           CUSTOMER
...
...
```

SET commands

SET commands enable you to view and change the **isql** environment.

SET AUTODDL

specifies whether DDL statements are committed automatically after being executed, or committed only after an explicit COMMIT.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Syntax

```
SQL> SET AUTODDL [ON | OFF] ; /* default is ON */
```

TABLE 10-19 SET AUTO[DDL] arguments

Argument	Description
ON	Toggles automatic commit on
OFF	Toggles automatic commit off

Shorthand

SET AUTO (with no argument) simply toggles AUTODDL on and off.

Example

```
...
SQL> SET AUTODDL OFF ;
SQL> CREATE TABLE WIZZO (x integer, y integer) ;
SQL> ROLLBACK; /* Table WIZZO is not created */
...
SQL> SET AUTO ON ;
SQL> CREATE TABLE WIZZO (x integer, y integer) ;
SQL> /* table WIZZO is created */
```

See Also *Firebird Reference Guide*—[SET AUTODDL](#) (ch. 1 p. 10)

SET BLOBDISPLAY

specifies both sub_type of BLOB to display and whether BLOB data should be displayed.

Syntax



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

SQL> SET BLOBDISPLAY [n | ALL | OFF] ;

TABLE 10-20 SET BLOB[DISPLAY] arguments

Argument	Description
n	BLOB SUB_TYPE to display. Default: <i>n</i> = 1 (text)
ALL	Display BLOB data of any sub_type
OFF	Toggles display of BLOB data off. The output shows only the Blob ID (two hex numbers separated by a colon (:). The first number is the ID of the table containing the BLOB column. The second is a sequenced instance number.

Shorthand SET BLOB is the same.

Example

```
...
SQL> SET BLOBDISPLAY OFF ;
SQL> SELECT PROJ_NAME, PROJ_DESC FROM PROJECT ;
SQL> /* rows show values for PROJ_NAME and Blob ID */
...
SQL> SET BLOB 1 ;
SQL> SELECT PROJ_NAME, PROJ_DESC FROM PROJECT ;
SQL> /* rows show values for PROJ_NAME and Blob ID */
SQL> /* and the blob text appears beneath each row */
```

SET COUNT

toggles off/on whether to display the number of rows retrieved by queries.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Syntax

```
SQL> SET COUNT [ON | OFF] ;
```

TABLE 10-21 SET COUNT arguments

Argument	Description
ON	Toggles on display of "rows returned" message
OFF	Toggles off display of "rows returned" message (default)

Example

```
...
SQL> SET COUNT ON ;
SQL> SELECT * FROM WIZZO WHERE FAVEFOOD = 'Pizza' ;
SQL> /* displays the data, followed by */

...
40 rows returned
```

See Also *Firebird Reference Guide*—[SET BLOBDISPLAY](#) (ch. 1 p. 12)

SET ECHO

toggles off/on whether commands are displayed before being executed. Default is ON but you might want to toggle it to OFF if sending your output to a script file.

Syntax



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

SQL> SET ECHO [ON | OFF] ; /* default is ON */

TABLE 10-22 SET ECHO arguments

Argument	Description
ON	Toggles on command echoing (default)
OFF	Toggles off command echoing

Example script wizzo.sql:

```
...
SET ECHO OFF ;
SELECT * FROM WIZZO WHERE FAVEFOOD = 'Pizza' ;
SET ECHO ON ;
SELECT * FROM WIZZO WHERE FAVEFOOD = 'Sardines' ;
EXIT;
...
SQL > INPUT wizzo.sql ;
```



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

```
WIZTYPE          FAVEFOOD
=====
alpha            Pizza
epsilon          Pizza
SELECT * FROM WIZZO WHERE FAVEFOOD = 'Sardines' ;
WIZTYPE          FAVEFOOD
=====
gamma            Sardines
lamda            Sardines
```

SET NAMES

specifies the character set that is to be active in database transactions.

Syntax

```
SQL> SET NAMES charset ;
```

TABLE 10-23 SET NAMES arguments

Argument	Description
charset	Name of the active character set. Default: NONE

Example in script:

```
...
SET NAMES ISO8859_1 ;
CONNECT 'phoenix:/usr/firebird/examples/employee.gdb' ;
```



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

SET PLAN

specifies whether to display the optimizer's query plan.

Syntax

```
SQL> SET PLAN [ ON | OFF ] ;
```

TABLE 10-24 SET PLAN arguments

Argument	Description
ON	Turns on display of the query plan. Default.
OFF	Turns off display of the query plan.

Shortcut: omit ON | OFF and use just SET PLAN as a toggle.

Example in script iscript.sql:

```
...
SET PLAN ON ;
SELECT JOB_COUNTRY, MIN_SALARY
FROM JOB
WHERE MIN_SALARY > 50000
AND JOB_COUNTRY = 'Sweden' ;
...
SQL> INPUT iscript.sql

PLAN (JOB INDEX (RDB$FOREIGN3,MINSALX,MAXSALX))
```

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

JOB_COUNTRY	MIN_SALARY
=====	=====
Sweden	120550.00

See Also *Firebird Reference Guide*—[SET PLAN](#) (ch. 1 p. 20)

SET PLANONLY

specifies to use the optimizer's query plan and display just the plan, without executing the actual query.
(Available in Firebird 1 and higher).

Syntax

```
SQL> SET PLANONLY ON | OFF;
```

The command works as a toggle switch. The argument is optional.

SET SQL DIALECT

specifies the Firebird SQL dialect to which the client session is to be changed. If the session is currently attached to a database of a different dialect to the one specified in the command, a warning is displayed.

Syntax

```
SQL> SET SQL DIALECT n ;
```

TABLE 10-25 SET SQL DIALECT arguments

Argument	Description
<i>n</i>	<i>n</i> = 1 for Dialect 1, 2 for Dialect 2, 3 for Dialect 3

Example SQL> SET SQL DIALECT 3 ;



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

SET STATS

specifies whether to display performance statistics following the output of a query.

Syntax

```
SQL> SET STATS [ ON | OFF ] ;
```

TABLE 10-26 SET STATS arguments

Argument	Description
ON	Turns on display of performance statistics. Displays <ul style="list-style-type: none">• Current memory available (bytes)• Change in available memory (bytes)• Maximum memory available (bytes)• Elapsed time for the operation (seconds)• CPU time for the operation (seconds)• Number of cache buffers used• Number of reads requested• Number of writes requested• Number of fetches done
OFF	Turns off display of performance statistics. Default.

Shortcut: omit ON | OFF and use just SET STATS as a toggle.

Example in script `script.sql`:

```
...
SET STATS ON ;
SELECT JOB_COUNTRY, MIN_SALARY
```



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

```
FROM JOB  
WHERE MIN_SALARY > 50000  
AND JOB_COUNTRY = 'Sweden';  
...  
SQL> INPUT iscript.sql
```

JOB_COUNTRY	MIN_SALARY
=====	=====
Sweden	120550.00

```
Current memory = 1234567  
Delta memory = 0  
Max memory = 191072220  
Elapsed time = 0.02 sec  
Buffers = 75  
Reads = 3  
Writes = 2  
Fetches = 441
```

See Also Firebird Reference Guide—[SET STATS](#) (ch. 1 p. 21)

SET TERM

specifies the character which will be used as the command or statement *terminator*, from the next statement forward.

Syntax

```
SQL> SET TERM string ;
```



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

TABLE 10-27 SET TERM arguments

Argument	Description
<i>string</i>	Character or characters which will be used as statement terminator. Default: ;

Example in script *iscript.sql*:

```
...
SET TERM ^^ ;
CREATE PROCEDURE ADD_WIZTYPE (WIZTYPE VARCHAR(16), FAVEFOOD VARCHAR(20))
AS
BEGIN
    BEGIN
        INSERT INTO WIZZO(WIZTYPE, FAVEFOOD)
        VALUES ( :WIZTYPE, :FAVEFOOD) ;
    END;
    RETURN;
END ^^
SET TERM ; ^^
...
```

See Also see [Statement terminator](#) on p. 501 of chapter 25

SET TIME

specifies whether to display the time portion of a DATE value (Dialect 1 only).

Syntax



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

SQL> SET TIME [ON | OFF] ;

TABLE 10-28 SET TIME arguments

Argument	Description
ON	Toggles on time portion display in Dialect 1 DATE value
OFF	Toggles on time portion display in Dialect 1 DATE value. Default.

Example SQL> SELECT HIRE_DATE FROM EMPLOYEE WHERE EMP_NO = 145;

```
HIRE_DATE
-----
16-MAY-2001
...
SQL> SET TIME ON ;
SQL> SELECT HIRE_DATE FROM EMPLOYEE WHERE EMP_NO = 145;
```

```
HIRE_DATE
-----
16-MAY-2001 18:20:00
```

SET WARNINGS

specifies whether warnings are to be output.

Syntax

```
SQL> SET WARNINGS [ ON | OFF ] ;
```



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

TABLE 10-29 SET WARNINGS arguments

Argument	Description
ON	Toggles on display of warnings if it was toggled off, or if the session was started with the -nowarnings option
OFF	Toggles off display of warnings if it is currently toggled on

Shorthand: SET WNG can be used as a substitute, as a simple on/off toggle.

Example SQL> SELECT HIRE_DATE FROM EMPLOYEE WHERE EMP_NO = 145;

HIRE_DATE

16-MAY-2001

...

SQL> SET TIME ON ;

SQL> SELECT HIRE_DATE FROM EMPLOYEE WHERE EMP_NO = 145;

HIRE_DATE

16-MAY-2001 18:20:00



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Extracting metadata

From the command line, you can use the **-extract** option to output the DDL statements that define the metadata for a database.

All reserved words and objects are extracted into the file in uppercase unless the local language uses a character set that has no upper case. The output script is created with **commit** following each set of commands, so that tables can be referenced in subsequent definitions. The output file includes the name of the object and the owner, if one is defined.

The optional **-output** flag reroutes output to a named file.

Use this syntax:

```
isql [[-extract | -x][-a] [[-output | -o] outputfile]] database;
```

The **-x** option can be used as an abbreviation for **-extract**. The **-a** flag directs **isql** to extract all database objects. Note that the output file specification, *outputfile*, must follow the **-output** flag. The name of the database being extracted can be at the end of the command.

You can use the resulting text file to:

- Examine the current state of a database's system tables before planning alterations. This is especially useful when the database has changed significantly since its creation.
- Create a database with schema definitions that are identical to the extracted database.
- Open in your text editor to make changes to the database definition or create a new database source file.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Examples using isql -extract

The following statement extracts the SQL schema from the database **employee.gdb** to a DDL script file called **employee.sql**:

```
isql -extract -output employee.sql employee.gdb;
```

This command is equivalent:

```
isql -x -output employee.sql employee.gdb;
```

This example combines the **-extract** and **-output** options to extract the DDL statements from the database **dev.gdb** into a file called **dev.out**. The output database name must follow the **-output** flag:

```
isql -extract -output dev.out dev.gdb
```

Using isql -a

The **-e(x)tract** option extracts metadata for SQL objects only. If you wish to extract a DDL script that includes other statements as well, use the **-a** option. It will extract the declarations to the database, such as **DECLARE EXTERNAL FUNCTION** and **DECLARE FILTER**.

UDF and blob filter code are not extracted, because they are not part of the database.

For example, to extract DDL statements from database **employee.gdb** and store in the file **employee.sql**, enter:

```
isql -a employee.gdb -output employee.sql
```

What isql [-e]x[tract] does not extract

- it does not extract UDF code and blob filters, because they are not part of the database.
- it does not extract system tables, system views, or system triggers.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- the extracted file does not show ownership, because DDL statements do not contain references to object ownership.
❖ There is no way to assign an object to its original owner.

★ To extract metadata in an interactive isql session, use the SHOW commands, optionally in combination with OUTPUT.

Exiting an interactive isql session

To exit the **isql** utility and **roll back** all uncommitted work, enter:

```
SQL> QUIT;
```

To exit the **isql** utility and **commit** all work, enter:

```
SQL> EXIT;
```

Command-line switches

Although **isql** is very handy as an interactive utility, you are not restricted to running it only in interactive mode.

- For many of its tasks, it can be invoked directly from the command shell of your operating system. In this mode, it executes the options for the switches specified in the command-line parameters and passes control back to the shell.
- Shell calls to **isql** can also be batched inside shell scripts or batch files.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Abbreviated commands

Only the initial characters in an option are required. You can also type any portion of the text enclosed in brackets, including the full option name. For example, specifying `-n`, `-no`, or `-noauto` have the same effect.

TABLE 10-30 isql command-line switches

Option	Description
<code>-a</code>	Extracts all DDL for the named database, including non-DDL statements
<code>-d[atabase] name</code>	Used with <code>-x</code> ; changes the CREATE DATABASE statement that is extracted to a file//Without <code>-d</code> , CREATE DATABASE appears as a C-style comment and uses the database name specified on the <code>isql</code> command line. With <code>-d</code> , <code>isql</code> extracts an uncommented CREATE DATABASE and substitutes <code>name</code> as its database argument
<code>-c[ache]</code>	Set number of cache buffers for this connection to the database. See <i>Default cache size per ISQL connection</i> .
<code>-e[cho]</code>	Displays (echoes) each statement before executing it
<code>-ex[tract]</code>	Extracts DDL for the named database; displays DDL to the screen unless redirected to a file
<code>-i[nput] file</code>	Reads commands from an input file instead of from standard input. Input files can contain <code>-input</code> commands that call other files, enabling execution to branch and then return. <code>isql</code> exits (with a commit) when it reaches the end of the first file. In interactive sessions, use <code>-input</code> to read commands from a file
<code>-m[erge_stderr]</code>	Merges <code>stderr</code> output with <code>stdout</code> . Useful for capturing output and errors to a single file when running <code>isql</code> in a shell script or batch file



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

TABLE 10-30 isql command-line switches

Option	Description
-n[oauto]	Turns off automatic commitment of DDL statements. By default, DDL statements are committed automatically in a separate transaction
-nowarnings	Displays warning messages if and only if an error occurs (by default, isql displays any message returned in a status vector, even if no error occurred).
-o[utput] file	Writes results to an output file instead of to standard output. In interactive sessions, use -output to write results to a file
-pas[sword] <i>password</i>	Used with -user //Specifies a password when connecting to a remote server. For access, both <i>password</i> and <i>user</i> must represent valid entries in the security database
-page[length] <i>n</i>	Prints column headers every <i>n</i> lines instead of the default 20
-q[uiet]	?
-r[ole] <i>rolename</i>	Grants privileges of role <i>rolename</i> to <i>user</i> on connection to the database
-s[ql_dialect] <i>n</i>	Interprets subsequent commands as dialect <i>n</i> until end of session or until dialect is changed by a SET SQL DIALECT statement. <ul style="list-style-type: none">• For <i>n</i> = 1, commands are processed as in InterBase 5• For <i>n</i> = 2, elements that have different interpretations in dialect 1 and 3 are all flagged with warnings or errors to assist in migrating databases to dialect 3• For <i>n</i> = 3, all statements are parsed with Firebird 1 SQL semantics: double quotes are delimited identifiers, DATE datatype is SQL DATE, and exact numerics with precision greater than 9 are stored as INT64
-t[erminator] <i>x</i>	Changes the end-of-statement symbol from the default semicolon (;) to <i>x</i> , where <i>x</i> is a single character or any sequence of characters



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

TABLE 10-30 isql command-line switches

Option	Description
<code>-u[ser] user</code>	Used with <code>-password</code> ; specifies a user name when connecting to a remote server. For access, both <i>password</i> and <i>user</i> must represent a valid entry in the security database
<code>-x</code>	Same as <code>-extract</code>
<code>-z</code>	Displays the software version of isql

[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

CHAPTER 11

Designing Databases

Defining data structures

Firebird is a relational database management system. As such, It is designed to support the creation and maintenance of abstract data structures not just to store data but also to maintain relationships and to optimize the speed and efficiency with which requested data can be returned to SQL client applications. This chapter covers the concepts, design and language for defining these structures.

Some essential terms

Data Definition Language (DDL)

The underlying structures of a database—its tables, views, and indexes—are created using a subset of the Firebird SQL language known as Data Definition Language (DDL). A DDL statement begins with one of the keywords CREATE, ALTER, RECREATE or DROP, causing a single object to be created, modified, reconstructed or destroyed, respectively. The database and, thereafter, its objects, rules and relationships form the structure of a relational database.

Metadata

Collectively, objects defined within a database are known as its *metadata* or, more traditionally, its *schema*. The process of creating, modifying, reconstructing and destroying metadata is referred to as *data definition*.

The system tables

Firebird stores metadata in a set of tables which it creates right inside the database—the system tables. All system tables have identifiers beginning with “RDB\$”. For example, the table which stores the definitions and other information about all of the table structures in your database is called RDB\$RELATIONS. A related table, RDB\$RELATION_FIELDS, stores information and definitions for the columns of each table.



Symbols **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

For full details of the system tables, refer to the *Firebird Reference Guide*—[System Tables and Views](#) (ch. 9 p. 346).

Design considerations

Description and analysis

A database abstractly represents a world of organization, relationships, rules and processes. Before reaching the point of being capable of designing the structures and rules for the database, the analyst-designer has much to do, working with people involved to identify the real structures, rules and requirements from which the database design will be rendered. The importance of scrupulous description and analysis can not be over-emphasized.

Logical database design is an iterative process of refining the structures derived from describing and analyzing the world whose scope is to be encompassed by the database. Large, heterogeneous structures of information are reduced progressively to smaller, more specialized data objects and are gradually mapped to a data model. An important part of this reduction process involves *normalization*—splitting out groups of data items with the goal of establishing pure relationships between distinct items, eliminating redundancies and associating connected items of data in structures that can be queried efficiently.

This phase can be one of the most challenging tasks for the database designer, especially in environments where the business has been attuned to operating with spreadsheets and desktop databases. Regrettably, even in established client/server environments, all too many poorly-performing, corruption-prone databases are found to have been “designed” using reports and spreadsheets as the basis for design.

A wealth of literature is available providing techniques for effective business problem analysis for relational database designers, a reflection of the maturity of our trade. For some recommendations, refer to the *Firebird Reference Guide*—[Resources and References](#) (ch. 10 p. 399).



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Data model <> database

The data model which evolves during description and analysis provides a logical blueprint for your data structures. It is rarely practicable to implement the logical model directly to the physical structures in which you will store data. Typically, for example, a logical model records many-to-many relationships which cannot be resolved physically into two tables.

A logical model is often several layers deep, expressing relationships and structures which will be implemented in your physical database using views, outputs calculated in stored procedures or in SQL itself. Dynamic structures for the selection and manipulation of data are the arteries of a client/server database. A database which directly implements even an excellent data model will lack flexibility and will under-perform if it does not take into account the power and economy of its dynamic capabilities.

The data model describes the logical structure of the world in its scope, including the data objects or entities, data types, user operations, relationships and, sometimes, integrity constraints. An effective database design implements the physical structures and capabilities that will assure not just the integrity of the logical model but also the effectiveness and efficiency of data access for users and administrators.

Some key points to consider in design include the following:

- A relational database is designed to output sets of data, not tables. Well-designed keys will guarantee that, if data exists in the database, it will be possible to access it in any logical context for which it is required.
- In a sound design, changes in physical structures should not affect logical structures.
- A common argument against normalization is that it proliferates tables. A strong counter-argument to that is that it avoids duplication of data (often inadvertent, because of clumsy design) and the untraceable integrity corruption that results.

Design goals

Although relational databases are very flexible, the only way to guarantee data integrity and satisfactory database performance is a solid database design—there is no built-in protection against poor design decisions. A good database design:

- **Satisfies the users' content requirements** for the database. Before you can design the database, you must do extensive research on the requirements of the users and how the database will be used.



Symbols **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

- Ensures the consistency and integrity of the data. When you design a table, you define certain attributes and constraints that restrict what a user or an application can enter into the table and its columns. By validating the data before it is stored in the table, the database enforces the rules of the data model and preserves data integrity.
- Provides a natural, easy-to-understand structuring of information. Good design makes queries easier to understand, so users are less likely to introduce inconsistencies into the data, or to be forced to enter redundant data. This facilitates database updates and maintenance.
- Satisfies the users' performance requirements. Good database design ensures better performance. If tables are allowed to be too large, or if there are too many (or too few) indexes, long waits can result. If the database is very large with a high volume of transactions, performance problems resulting from poor design are magnified.

Design framework

The following steps provide a framework for designing a database:

- 1 Determine the information requirements for the database by interviewing prospective users.
- 2 Analyze the real-world objects that you want to model in your database. Organize the objects into entities and attributes and make a list.
- 3 Map the entities and attributes to Firebird tables and columns.
- 4 Determine an attribute that will uniquely identify each object.
- 5 Develop a set of rules that govern how each table is accessed, populated, and modified.
- 6 Establish relationships between the objects (tables and columns).
- 7 Plan database security.

The following sections describe each of these steps in more detail.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Analyzing requirements

The first step in the design process is to research the environment that you are trying to model. This involves interviewing prospective users in order to understand and document their requirements. Ask the following types of questions:

- Will your applications continue to function properly during the implementation phase? Will the system accommodate existing applications, or will you need to restructure applications to fit the new system?
- Whose applications use which data? Will your applications share common data?
- How do the applications use the data stored in the database? Who will be entering the data, and in what form? How often will the data objects be changed?
- What access do current applications require? Do your applications use only one database, or do they need to use several databases which might be different in structure? What access do they anticipate for future applications, and how easy is it to implement new access paths?
- Which information is the most time-critical, requiring fast retrieval or updates?

Collecting and analyzing data

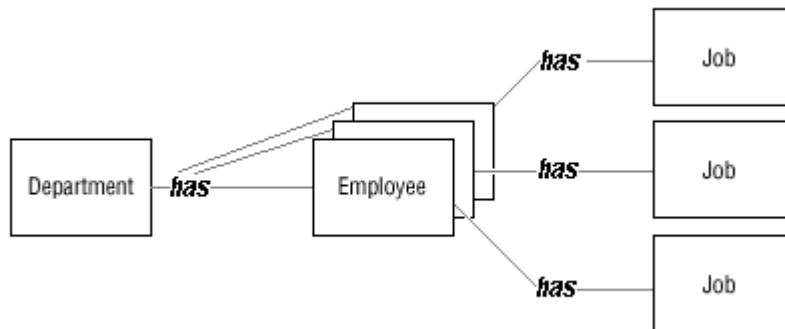
Before designing the database objects—the tables and columns—you need to organize and analyze the real-world data on a conceptual level. There are four primary goals:

- **Identify the major functions and activities of your organization.** For example: hiring employees, shipping products, ordering parts, processing paychecks, and so on.
- **Identify the objects of those functions and activities.** Building a business operation or transaction into a sequence of events will help you identify all of the entities and relationships the operation entails. For example, when you look at a process like “hiring employees,” you can immediately identify entities such as the JOB, the EMPLOYEE, and the DEPARTMENT.
- **Identify the characteristics of those objects.** For example, the EMPLOYEE entity might include such information as EMPLOYEE_ID, FIRST_NAME, LAST_NAME, JOB, SALARY, and so on.
- **Identify certain relationships between the objects** For example, how do the EMPLOYEE, JOB, and DEPARTMENT entities relate to each other? The employee has one job title and belongs to one department, while a single department has many employees and jobs. Simple graphical flow charts help to identify the relationships.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

FIGURE 1 Identifying relationships between objects



Identifying entities and attributes

Based on the requirements that you collect, identify the objects that need to be in the database—the entities and attributes. An *entity* is a type of person, object, or thing that needs to be described in the database. It might be an object with a physical existence, like a person, a car, or an employee, or it might be an object with a conceptual existence, like a company, a job, or a project. Each entity has properties, called *attributes*, that describe it.

Suppose you are designing a database that must contain information about each employee in the company, departmental-level information, information about current projects, and information about customers and sales. The example below shows how to create a list of entities and attributes that organizes the required data.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

TABLE 11–2 List of entities and attributes

Entities	Attributes
EMPLOYEE	Employee Number Last Name First Name Department Number Job Code Phone Extension Salary
DEPARTMENT	Department Number Department Name Department Head Name Department Head Employee Number Budget Location Phone Number
PROJECT	Project ID Project Name Project Description



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

TABLE 11-2 List of entities and attributes (*continued*)

Entities	Attributes
	Team Leader
	Product
CUSTOMER	Customer Number
	Customer Name
	Contact Name
	Phone Number
	Address
SALES	PO Number
	Customer Number
	Sales Rep
	Order Date
	Ship Date
	Order Status

By listing the entities and associated attributes this way, you can begin to eliminate redundant entries. Do the entities in your list work as tables? Should some columns be moved from one group to another? Does the same attribute appear in several entities? Each attribute should appear only once, and you need to determine which entity is the primary owner of the attribute. For example, DEPARTMENT HEAD NAME should be eliminated because employee names (FIRST NAME and LAST NAME) already exist in the EMPLOYEE entity. DEPARTMENT HEAD EMPLOYEE NUM can then be used to access all of the employee-specific information by referencing

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

EMPLOYEE NUMBER in the EMPLOYEE entity. For more information about accessing information by reference, see [Establishing relationships between objects](#).

The next section describes how to map your lists to actual database objects—entities to tables and attributes to columns.

Designing tables

In a relational database, the database object that represents a single entity is a *table*, which is a two-dimensional matrix of rows and columns. Each column in a table represents an attribute. Each row in the table represents a specific *instance* of the entity. After you identify the entities and attributes, create the *data model*, which serves as a logical design framework for creating your Firebird database. The data model maps entities and attributes to Firebird tables and columns, and is a detailed description of the database—the tables, the columns, the properties of the columns, and the relationships between tables and columns.

The example below shows how the EMPLOYEE entity from the entities/attributes list has been converted to a table.

TABLE 11-3 EMPLOYEE table

EMP_NO	LAST_NAME	FIRST_NAME	DEPT_NO	JOB_CODE	PHONE_EXT	SALARY
24	Smith	John	100	Eng	4968	64000
48	Carter	Catherine	900	Sales	4967	72500
36	Smith	Jane	600	Admin	4800	37500

Each row in the EMPLOYEE table represents a single employee. EMP_NO, LAST_NAME, FIRST_NAME, DEPT_NO, JOB_CODE, PHONE_EXT, and SALARY are the columns that represent employee attributes. When the table is populated with data, rows are added to the table, and a *value* is stored at the intersection of each row and column, called a field. In the EMPLOYEE table, “Smith” is a data value that resides in a single field of an employee record.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Determining unique attributes

One of the tasks of database design is to provide a way to uniquely identify each occurrence or instance of an entity so that the system can retrieve any single row in a table. The values specified in the table's primary key distinguish the rows from each other. A PRIMARY KEY or UNIQUE constraint ensures that values entered into the column or set of columns are unique in each row. If you try to insert a value in a PRIMARY KEY or UNIQUE column that already exists in another row of the same column, Firebird prevents the operation and returns an error.

For example, in the EMPLOYEE table, EMP_NO is a unique attribute that can be used to identify each employee in the database, so it is the primary key. When you choose a value as a primary key, determine whether it is inherently unique. For example, no two social security numbers or driver's license numbers are ever the same. Conversely, you should not choose a name column as a unique identifier due to the probability of duplicate values. If no single column has this property of being inherently unique, then define the primary key as a composite of two or more columns which, when taken together, are unique.

A unique key is different from a primary key in that a unique key is not the primary identifier for the row, and is not typically referenced by a foreign key in another table. The main purpose of a unique key is to force a unique value to be entered into the column. You can have only one primary key defined for a table, but any number of unique keys.

Developing a set of rules

When designing a table, you need to develop a set of rules for each table and column that establishes and enforces data integrity. These rules include:

- Specifying a data type
- Choosing international character sets
- Creating a domain-based column
- Setting default values and NULL status
- Defining integrity constraints and cascading rules
- Defining CHECK constraints



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Specifying a data type

Once you have chosen a given attribute as a column in the table, you must choose a data type for the attribute. The data type defines the set of valid data that the column can contain. The data type also determines which operations can be performed on the data, and defines the disk space requirements for each data item.

The general categories of SQL data types include:

- Character data types
- Whole number (integer) data types
- Fixed and floating decimal data types
- data types for dates and times
- A blob data type to represent data of unspecified length and structure, such as such as graphics and digitized voice; blobs can be numeric, text, or binary

 For more information about data types supported by Firebird, see the chapters on [Tables](#) and [Firebird Data Types](#).

Choosing international character sets

When you create the database, you can specify a default character set. A default character set determines:

- What characters can be used in CHAR, VARCHAR, and BLOB text columns.
- The default collation order used in sorting a column.

The *collation order* determines the order in which values are sorted. The COLLATE clause of CREATE TABLE allows users to specify a particular collation order for columns defined as CHAR and VARCHAR text data types. You must choose a collation order that is supported for the column's given character set. The collation order set at the column level overrides a collation order set at the domain level.

Choosing a default character set is intended primarily for users who are interested in providing a database for international use. For example, the following statement creates a database that uses the ISO8859_1 character set, typically used to support European languages:

```
CREATE DATABASE 'employee.gdb'  
DEFAULT CHARACTER SET ISO8859_1;
```

You can override the database default character set by creating a different character set for a column when specifying the data type. The data type specification for a CHAR, VARCHAR, or BLOB text column definition can include a CHARACTER SET clause to specify a particular character set for a column. If you do not specify a

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

character set, the column assumes the default database character set. If the database default character set is subsequently changed, all columns defined after the change have the new character set, but existing columns are not affected.

If you do not specify a default character set at the time the database is created, the character set defaults to NONE. This means that there is no character set assumption for the columns; data is stored and retrieved just as it was originally entered. You can load any character set into a column defined with NONE, but you cannot load that same data into another column that has been defined with a different character set. No transliteration will be performed between the source and the destination character sets.



For a list of the international character sets and collation orders that Firebird supports, *Firebird Reference Guide*—[Character Sets and Collation Orders](#) (ch. 4 p. 249). For discussion of the issues, see [Character Sets and Collation Orders](#) on p. 301 of chapter 16 in this volume.

Specifying domains

When several tables in the database contain columns with the same definitions and data types, you can create domain definitions and store them in the database. Users who create tables can then reference the domain definition to define column attributes locally.



For more information about creating and referencing domains, see [Firebird domains](#) on page 285 in chapter 15.

Setting default values and NULL status

When you define a column, you have the option of setting a DEFAULT value. This value is used whenever an INSERT or UPDATE on the table does not supply an explicit value for the column. Defaults can save data entry time and prevent data entry errors. For example, a possible default for a DATE column could be today's date; in a Y/N flag column for saving changes, "Y" could be the default. Column-level defaults override defaults set at the domain level. Some examples:

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

```
stringfld VARCHAR(10) DEFAULT 'abc'  
integerfld INTEGER DEFAULT 1  
numfld NUMERIC(15,4) DEFAULT 1.5  
datefld1 DATE DEFAULT '5/20/2000'  
datefld2 DATE DEFAULT 'TODAY'  
userfld VARCHAR(12) DEFAULT USER
```

The last two show special Firebird features: 'TODAY' defaults to the current date, and USER is the user who is performing an insert to the column.

Assign a NULL default to insert a NULL into the column if the user does not enter a value. Assign NOT NULL to force the user to enter a value, or to define a default value for the column. NOT NULL must be defined for PRIMARY KEY and UNIQUE key columns.

Defining integrity constraints

Integrity constraints are rules that govern column-to-table and table-to-table relationships, and validate data entries. They span all transactions that access the database and are maintained automatically by the system. Integrity constraints can be applied to an entire table or to an individual column. A PRIMARY KEY or UNIQUE constraint guarantees that no two values in a column or set of columns are the same.

Data values that uniquely identify rows (a primary key) in one table can also appear in other tables. A foreign key is a column or set of columns in one table that contain values that match a primary key in another table. The ON UPDATE and ON DELETE referential constraints allow you to specify what happens to the referencing foreign key when the primary key changes or is deleted.



For more information on using PRIMARY KEY and FOREIGN KEY constraints, see the chapter [Tables](#).

Defining CHECK constraints

Along with preventing the duplication of values using UNIQUE and PRIMARY KEY constraints, you can specify another type of data entry validation. A CHECK constraint places a condition or requirement on the data values in a column at the time the data is entered. The CHECK constraint enforces a search condition that must be true in order to insert into or update the table or column.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Establishing relationships between objects

The relationship between tables and columns in the database must be defined in the design. For example, how are employees and departments related? An employee can have only one department (a one-to-one relationship), but a department has many employees (a one-to-many relationship). How are projects and employees related? An employee can be working on more than one project, and a project can include several employees (a many-to-many relationship). Each of these different types of relationships has to be modeled in the database.

The relational model represents one-to-many relationships with primary key/foreign key pairings. Refer to the following two tables. A project can include many employees, so to avoid duplication of employee data, the PROJECT table can reference employee information with a foreign key. TEAM_LEADER is a foreign key referencing the primary key, EMP_NO, in the EMPLOYEE table.

TABLE 11-4 PROJECT table

PROJ_ID	TEAM_LEADER	PROJ_NAME	PROJ_DESC	PRODUCT
DGPII	44	Automap	blob data	hardware
VBASE	47	Video database	blob data	software
HWRII	24	Translator upgrade	blob data	software

TABLE 11-5 EMPLOYEE table

EMP_NO	LAST_NAME	FIRST_NAME	DEPT_NO	JOB_CODE	PHONE_EXT	SALARY
24	Smith	John	100	Eng	4968	64000
48	Carter	Catherine	900	Sales	4967	72500
36	Smith	Jane	600	Admin	4800	37500



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

For more information on using PRIMARY KEY and FOREIGN KEY constraints, see the chapter [Tables](#).

Enforcing referential integrity

The primary reason for defining foreign keys is to ensure that the integrity of the data is maintained when more than one table references the same data—rows in one table must always have corresponding rows in the referencing table. Firebird enforces *referential integrity* in the following ways:

- Before a foreign key can be added, the unique or primary keys that the foreign key references must already be defined.
- If information is changed in one place, it must be changed in every other place that it appears. Firebird does this automatically when you use the ON UPDATE option to the REFERENCES clause when defining the constraints for a table or its columns. You can specify that the foreign key value be changed to match the new primary key value (CASCADE), or that it be set to the column default (SET DEFAULT), or to null (SET NULL). If you choose NO ACTION as the ON UPDATE action, you must manually ensure that the foreign key is updated when the primary key changes. For example, to change a value in the EMP_NO column of the EMPLOYEE table (the primary key), that value must also be updated in the TEAM_LEADER column of the PROJECT table (the foreign key).
- When a row containing a primary key in one table is deleted, the meaning of any rows in another table that contain that value as a foreign key is lost unless appropriate action is taken. Firebird provides the ON DELETE option to the REFERENCES clause of CREATE TABLE and ALTER TABLE so that you can specify whether the foreign key is deleted, set to the column default, or set to null when the primary key is deleted. If you choose NO ACTION as the ON DELETE action, you must manually delete the foreign key before deleting the referenced primary key.
- Firebird also prevents users from adding a value in a column defined as a foreign key that does not reference an existing primary key value. For example, to change a value in the TEAM_LEADER column of the PROJECT table, that value must first be updated in the EMP_NO column of the EMPLOYEE table.

For more information on using PRIMARY KEY and FOREIGN KEY constraints, see [Tables](#) on page 313 in chapter 17.



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

Normalizing the database

After your tables, columns, and keys are defined, look at the design as a whole and analyze it using normalization guidelines in order to find logical errors. As mentioned in the overview, normalization involves breaking down larger tables into smaller ones in order to group data together that is naturally related.

When a database is designed using proper normalization methods, data related to other data does not need to be stored in more than one place—if the relationship is properly specified. The advantages of storing the data in one place are:

- The data is easier to update or delete.
- When each data item is stored in one location and accessed by reference, the possibility for error due to the existence of duplicates is reduced.
- Because the data is stored only once, the possibility for introducing inconsistent data is reduced.

In general, the normalization process includes:

- Eliminating repeating groups.
- Removing partly-dependent columns.
- Removing transitively-dependent columns.

An explanation of each step follows.

Eliminating repeating groups

When a field in a given row contains more than one value for each occurrence of the primary key, then that group of data items is called a *repeating group*. This is a violation of the first normal form, which does not allow multi-valued attributes.

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

Refer to the DEPARTMENT table. For any occurrence of a given primary key, if a column can have more than one value, then this set of values is a repeating group. Therefore, the first row, where DEPT_NO = "100", contains a repeating group in the DEPT_LOCATIONS column.

TABLE 11-6 DEPARTMENT table

DEPT_NO	DEPARTMENT	HEAD_DEPT	BUDGET	DEPT_LOCATIONS
100	Sales	000	1000000	Monterey, Santa Cruz, Salinas
600	Engineering	120	1100000	San Francisco
900	Finance	000	400000	Monterey

In the next example, even if you change the attribute to represent only one location, for every occurrence of the primary key "100", all of the columns contain repeating information *except* for DEPT_LOCATION, so this is still a repeating group.

TABLE 11-7 DEPARTMENT table

DEPT_NO	DEPARTMENT	HEAD_DEPT	BUDGET	DEPT_LOCATION
100	Sales	000	1000000	Monterey
100	Sales	000	1000000	Santa Cruz
600	Engineering	120	1100000	San Francisco
100	Sales	000	1000000	Salinas

To normalize this table, we could eliminate the DEPT_LOCATION attribute from the DEPARTMENT table, and create another table called DEPT_LOCATIONS. We could then create a primary key that is a combination of

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

DEPT_NO and DEPT_LOCATION. Now a distinct row exists for each location of the department, and we have eliminated the repeating groups.

TABLE 11-8 DEPT_LOCATIONS table

DEPT_NO	DEPT_LOCATION
100	Monterey
100	Santa Cruz
600	San Francisco
100	Salinas

Removing partly-dependent columns

Another important step in the normalization process is to remove any non-key columns that are dependent on only part of a composite key. Such columns are said to have a *partial key dependency*. Non-key columns provide information about the subject, but do not uniquely define it.

For example, suppose you wanted to locate an employee by project, and you created the PROJECT table with a composite primary key of EMP_NO and PROJ_ID.

TABLE 11-9 PROJECT table

EMP_NO	PROJ_ID	LAST_NAME	PROJ_NAME	PROJ_DESC	PRODUCT
44	DGPII	Smith	Automap	blob data	hardware
47	VBASE	Jenner	Video database	blob data	software
24	HWRII	Stevens	Translator upgrade	blob data	software

The problem with this table is that PROJ_NAME, PROJ_DESC, and PRODUCT are attributes of PROJ_ID, but not EMP_NO, and are therefore only partly dependent on the EMP_NO/PROJ_ID primary key. This is also true for LAST_NAME because it is an attribute of EMP_NO, but does not relate to PROJ_ID. To normalize this table, we would remove the EMP_NO and LAST_NAME columns from the PROJECT table, and create another table called

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

EMPLOYEE_PROJECT that has EMP_NO and PROJ_ID as a composite primary key. Now a unique row exists for every project that an employee is assigned to.

Removing transitively-dependent columns

The third step in the normalization process is to remove any non-key columns that depend upon other non-key columns. Each non-key column must be a fact about the primary key column. For example, suppose we added TEAM_LEADER_ID and PHONE_EXT to the PROJECT table, and made PROJ_ID the primary key. PHONE_EXT is a fact about TEAM_LEADER_ID, a non-key column, not about PROJ_ID, the primary key column:

TABLE 11-10 PROJECT table

PROJ_ID	TEAM_LEADER_ID	PHONE_EXT	PROJ_NAME	PROJ_DESC	PRODUCT
DGPII	44	4929	Automap	blob data	hardware
VBASE	47	4967	Video database	blob data	software
HWRII	24	4668	Translator upgrade	blob data	software

To normalize this table, we would remove PHONE_EXT, change TEAM_LEADER_ID to TEAM_LEADER, and make TEAM_LEADER a foreign key referencing EMP_NO in the EMPLOYEE table.

TABLE 11-11 PROJECT table

PROJ_ID	TEAM_LEADER	PROJ_NAME	PROJ_DESC	PRODUCT
DGPII	44	Automap	blob data	hardware
VBASE	47	Video database	blob data	software
HWRII	24	Translator upgrade	blob data	software

**Symbols** [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

TABLE 11-12 EMPLOYEE table

EMP_NO	LAST_NAME	FIRST_NAME	DEPT_NO	JOB_CODE	PHONE_EXT	SALARY
24	Smith	John	100	Eng	4968	64000
48	Carter	Catherine	900	Sales	4967	72500
36	Smith	Jane	600	Admin	4800	37500

When to break the rules

You should try to correct any normalization violations, or else make a conscious decision to ignore them in the interest of ease of use or performance. Just be sure that you understand the design trade-offs that you are making, and document your reasons. It might take several iterations to reach a design that is a desirable compromise between purity and reality, but this is the heart of the design process.

For example, suppose you always want data about dependents every time you look up an employee, so you decide to include DEP1_NAME, DEP1_BIRTHDATE, and so on for DEP1 through DEP30, in the EMPLOYEE table.

Generally speaking, that is terrible design, but the requirements of your application are more important than the abstract purity of your design. In this case, if you wanted to compute the average age of a given employee's dependents, you would have to explicitly add field values together, rather than asking for a simple average. If you wanted to find all employees with a dependent named "Jennifer," you would have to test 30 fields for each employee instead of one. If those are not operations that you intend to perform, then go ahead and break the rules. If the efficiency attracts you less than the simplicity, you might consider defining a view that combines records from employees with records from a separate DEPENDENTS table.

While you are normalizing your data, remember that Firebird offers direct support for array columns, so if your data includes, for example, hourly temperatures for twenty cities for a year, you could define a table with a character column that contains the city name, and a 24 by 366 matrix to hold all of the temperature data for one city for one year. This would result in a table containing 20 rows (one for each city) and two columns, one NAME column and one TEMP_ARRAY column. A normalized version of that record might have 366 rows per city, each of which would hold a city name, a Julian date, and 24 columns to hold the hourly temperatures.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Choosing indexes

Once you have your design, you need to consider what indexes are necessary. The basic trade-off with indexes is that more distinct indexes make retrieval by specific criteria faster, but updating and storage slower. One optimization is to avoid creating several indexes on the same column. For example, if you sometimes retrieve employees based on name, department, badge number, or department name, you should define one index for each of these columns. If a query includes more than one column value to retrieve, Firebird will use more than one index to qualify records. In contrast, defining indexes for every permutation of those three columns will actually slow both retrieval and update operations.

When you are testing your design to find the optimum combination of indexes, remember that the size of the tables affects the retrieval performance significantly. If you expect to have tables with 10,000 to 100,000 records each, do not run tests with only 10 to 100 records.

Another factor that affects index and data retrieval times is page size. By increasing the page size, you can store more records on each page, thus reducing the number of pages used by indexes. If any of your indexes are more than 4 levels deep, you should consider increasing the page size. If indexes on volatile data (data that is regularly deleted and restored, or data that has index key values that change frequently) are less than three levels deep, you should consider reducing your page size. In general, you should use a page size larger than your largest record, although Firebird's data compression will generally shrink records that contain lots of string data, or lots of numeric values that are 0 or NULL. If your records have those characteristics, you can probably store records on pages which are 20% smaller than the full record size. On the other hand, if your records are not compressible, you should add 5% to the actual record size when comparing it to the page size.



For more information on creating indexes, see [Indexes](#) on page 352 in chapter 18.

Increasing cache size

When Firebird reads a page from the database onto disk, it stores that page in its cache, which is a set of buffers that are reserved for holding database pages. Ordinarily, the default cache size of 2,048 buffers is adequate. If your application includes joins of five or more tables, Firebird automatically increases the size of the cache. If your application is well localized, that is, it uses the same small part of the database repeatedly, you might want to consider increasing the cache size so that you never have to release one page from cache to make room for another.

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

You can use the `gfix` utility to increase the default number of buffers for a specific database using the following command:

```
gfix -buffers n database_name
```

You can also change the default cache size for an entire server either by setting the value of `DATABASE_CACHE_PAGES` in the configuration file or by changing it on the IB Settings page of the Firebird Server Properties dialog on Windows platforms. This setting is not recommended because it affects all databases on the server and can easily result in overuse of memory or in unusably small caches. It's better to tune your cache on a per-database basis using `gfix -buffers`.

For more information about cache size, see [Configuring the database cache](#) on page 67 in chapter 5.

Creating a multi-file, distributed database

Multi-file databases were designed to avoid confining databases to the size of a single disk on systems that do not support spanning files across multiple disks. There will be no problems installing a RAID array and distributing a multi-file Firebird database across several disks on any supported platform.

Planning security

Planning security for a database is important. When implementing the database design, you should answer the following questions:

- Who will have authority to use Firebird?
- Who will have authority to open a particular database?
- Who will have authority to create and access a particular database object within a given database?

 For information about implementing security in Firebird, refer to the chapter [Managing Security](#).

Naming objects

Valid names for Firebird objects, such as tables, normally must have the following characteristics:

- consist of ASCII characters
- may not contain spaces
- are not case sensitive

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

- cannot be Firebird reserved words

In a dialect 3 database you *can* give an Firebird object a name that violates one or more of those rules—provided you always surround the name with double quotes. Such names are called *delimited identifiers*. When you use a double-quote delimited identifier, the name is case sensitive.

- ★ When you use an object name *without* double quotes, Firebird stores the identifier in upper-case but treats its identifier case-insensitively. If you create a table with a double-quote delimited name in all upper-case, you can use the name subsequently without double quotes, e.g.

```
CREATE TABLE "upper-case_NAME" ...  
SELECT * FROM upper-case_NAME;
```



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

CHAPTER 12

Writing and Running Scripts

With Firebird, as with all true SQL database management systems, you build your database and its objects—the *metadata* or *schema* of a database—using statements from a specialized subset of SQL statements known as *Data Definition Language*, or DDL. A batch of DDL statements in a text file is known as a *script*. A script, or a set of scripts, can be processed by `isql` directly at the command line or through a tool that provides a user-friendly interface for `isql`'s script processing capability.



For a list of such tools, refer to *Firebird Reference Guide*—[Resources and References](#) (ch. 10 p. 399).

About Firebird scripts

A script for creating and altering database objects is sometimes referred to as a *data definition file* or, more commonly, a *DDL script*. A DDL script can contain certain `isql` statements, specifically some of the `SET <parameter>` commands. `COMMIT` is also a valid statement in a script.

Other scripts can be written for inserting basic or “control” data into tables, updating columns, performing data conversions and other maintenance tasks involving data manipulation. These are known as DML scripts (for *Data Manipulation Language*).

DDL and DML commands can be mixed in a script. Because one of the important tasks of the interactive SQL tool `isql` is to process scripts, several `isql` commands are also accepted in scripts.

Script processing allows “chaining” of scripts, linking one script file to another by means of the `isql INPUT <filespec>` statement.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Script statements are executed in strict order. Use of the SET AUTODDL command enables you to control where statements or blocks of statements will be committed. It is also an option to defer committing the contents of a script until the entire script has been processed.

Why use scripts?

It is very good practice to use DDL scripts to create your database and its objects. Some of the reasons include

- self-documentation. A script is a plain text file, easily handled in any development system, both for updating and reference. Scripts can—and should—include detailed comment text. Alterations to metadata can be signed and dated manually.
- control of database development. Scripting all database definitions allows schema creation to be integrated closely with design tasks and code review cycles.
- repeatable and traceable creation of metadata. A completely reconstructable schema is a requirement in the quality assurance and disaster recovery systems of many organizations.
- orderly construction and reconstruction of database metadata. Experienced Firebird programmers often create a set of DDL scripts, designed to run and commit in a specific order, to make debugging easy and ensure that objects will exist when later, dependent objects refer to them.

How to create scripts

You can create DDL scripts in several ways, including:

- in an interactive `isql` session using the OUTPUT command to pass a series of DDL statements to a file
 For more information, see chapter 10, [Interactive SQL Utility \(isql\)](#) (p. 152), especially the OUTPUT and INPUT commands
- in a plain ASCII text editor that formats line breaks according to the rules of the operating system shell in which the DDL script will be executed

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

- using one of the many specialized script editor tools that are available in third-party admin tools for Firebird, such as Marathon, IB_SQL or InterBase Workbench
- using a CASE tool which can output DDL scripts according to the Firebird (InterBase) conventions

Executing scripts

You can execute DDL scripts in an interactive `isql` session using the INPUT command. Many of the third-party tools mentioned in the *Firebird Reference Guide*— [Resources and References](#) (ch. 10 p. 399) have the ability to execute and even to intelligently debug scripts in a GUI environment.

Basic steps

The basic steps for using script files are:

- 1 Create the script file using a text editor.

★ At the learning stage, you might wish to follow each DDL statement with a COMMIT statement, to ensure that an object will be visible to subsequent statements. As you become more practised, you will learn to commit statements in blocks, employing SET AUTODDL ON and SET AUTODDL OFF as a means of controlling interdependencies and testing/debugging scripts.

- 2 Execute the script using the INPUT command in an `isql` session or the Execute button (or equivalent) in your database management tool.
- 3 View output and confirm database changes.

★ Tools and Firebird `isql` versions vary in the information they return when a script trips up on a bad command. A feature added after Firebird 1.0 provides better script error reporting than previous versions.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Creating a SQL script

You can use any text editor to create a SQL script file, as long as the final file format is plain text (ASCII) and has lines terminated according to the rules of your operating system shell:

- on Windows the line terminator is carriage return + line feed (ASCII 13 followed by ASCII 10)
- on Linux/UNIX it is line feed or "new line" (ASCII 10)
- on Mac OSX it is newline (ASCII 10) and on native Macintosh it is carriage return (ASCII 13)

❖ Some editing tools provide the capability to save in different text formats. It may prove useful to be able to save Linux-compatible scripts on a Windows machine, for example. However, take care that you use an editor that saves only plain ASCII text.

Every SQL script file must begin with either a CREATE DATABASE statement or a CONNECT statement (including username and password in single quotes) that specifies the database on which the script file is to operate. The CONNECT or CREATE statement must contain a complete database file name and directory path.

For example:

```
/* SET SQL DIALECT 3 ; */  
CREATE DATABASE 'd:\databases\MyDatabase.gdb' PAGE_SIZE 8192 USER  
'SYSDBA' PASSWORD 'masterkey' ;
```

or

```
CONNECT 'd:\databases\MyDatabase.gdb' USER 'SYSDBA' PASSWORD 'masterkey' ;
```

❖ You cannot set dialect in a CREATE DATABASE statement.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

What is in a DDL script?

SQL statements

A DDL script consists of one or more SQL statements. As well as DDL statements to CREATE, ALTER, or DROP a database or any other object, it can include data manipulation language (DML) statements. Procedure language (PSQL) statements defining stored procedures and triggers can also be included. PSQL blocks get special treatment in scripts with regard to statement terminator symbols (see below).

- ★ It is quite common for INSERT statements to appear in DDL scripts, to populate some tables with control data. You might, for example, include statements to insert the initial rows in a table of Account Types.

Comments

A script can also contain comments, in two varieties.

Block comments

Block comments in DDL scripts use the C convention:

```
/* This comment can span multiple
lines in a script*/
```

A block comment can occur on the same line as a SQL statement or `isql` command and can be of any length, as long as it is preceded by `/*` and followed by `*/`.

In-line comments

The `/* . . . */` style of comment can also be embedded inside a statement as an in-line comment, e.g.

**Symbols** [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

```
CREATE TABLE USERS1 (
    USER_NAME            VARCHAR( 128 ) /* security user name */
, GROUP_NAME           VARCHAR( 128 ) /* not used on Windows */
, PASSWD               VARCHAR( 32 )  /* will be stored encrypted */
, FIRST_NAME           VARCHAR( 96 )   /* Defaulted */
, MIDDLE_NAME          VARCHAR( 96 )   /* Defaulted */
, LAST_NAME             VARCHAR( 96 )   /* Defaulted */
, FULL_NAME            VARCHAR( 290 )  /* Computed */
)
```

One-line comments

In Firebird scripts you can use an alternative convention for commenting a single line: the double hyphen, e.g.

```
-- comment
```

❖ This double-hyphen style of comment can *not* be used for in-line comments, even if you split a statement across several lines as in the CREATE TABLE example above.

isql statements

The **isql** commands SET AUTODDL, SET SQL DIALECT, SET TERM and INPUT are valid statements in a Firebird script—see chapter 10, [Interactive SQL Utility \(isql\)](#) (p. 152) for details of these commands.

Terminator symbols

All statements that are to be executed in the script must end with a terminator symbol.

- The default symbol is the semicolon (:).
- The default terminator can be overridden for all statements *except procedure language statements* (PSQL) by issuing a SET TERM command in the script.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Terminators and procedure language (PSQL)

PSQL does not permit any terminator other than the default semi-colon (:). This restriction is necessary because CREATE PROCEDURE, RECREATE PROCEDURE, ALTER PROCEDURE, CREATE TRIGGER and ALTER TRIGGER, together with their subsequent PSOL statements, are complex statements in their own right—statements within a statement. The compiler needs to see semi-colons in order to recognize each distinct PSQL statement.

Thus, in scripts, it is necessary to override the terminator being used for script commands before commencing to issue the PSQL statements for a stored procedure or a trigger. After the last END statement of the procedure source has been terminated, the terminator should be reset to the default using another SET TERM statement.

Examples

```
...
CREATE GENERATOR GEN_MY_GEN ;
SET TERM ^^ ;
CREATE TRIGGER BI_TABLEA_0 FOR TABLEA
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
  IF (NEW.PK IS NOT NULL) THEN
    NEW.PK = GEN_ID(GEN_MY_GEN, 1);
END ^^
SET TERM ; ^^
...
```

Any string may be used as an alternative terminator, for example:

```
...
```



CHAPTER 12 Writing and Running Scripts } What is in a DDL script?

Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

```
SET TERM @!# ;
CREATE PROCEDURE...
AS
BEGIN
  ... ;
  ... ;
END @!#
SET TERM ; @!#
/**/
COMMIT;
/**/

SET TERM + ;
CREATE PROCEDURE...
AS
BEGIN
  ... ;
  ... ;
END +
SET TERM ; +
/**/
COMMIT;
```

- ❖ The SQL statement silently fails if significant text follows the terminator character on the same line. Whitespace and comments can safely follow the terminator, but other statements cannot.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Committing Statements in Scripts

DDL statements

Statements in DDL scripts can be committed in one or more of the following ways:

- by including COMMIT statements at appropriate points in the script to ensure that new database objects are available to all subsequent statements that depend on them.

- by including this statement at the beginning of the script:

```
SET AUTODDL ON ;
```

To turn off automatic commit of DDL in an isql script, use SET AUTODDL OFF.

★ The ON and OFF keywords are optional: SET AUTODDL is a switch. For clarity of self-documentation, it is recommended that you include ON and OFF when using SET AUTODDL

- if you are running your script in isql, changes to the database from data definition (DDL) statements—for example, CREATE and ALTER statements—are automatically committed by default. This means that other users of the database see changes as soon as each DDL statement is executed.

❖ Some scripting tools deliberately turn off this autocommitting behavior when running scripts, since it can make debugging difficult. Make sure you understand the behavior of any third-party tool you use for scripts.

DML statements

Changes made to the database by data manipulation (DML) statements—INSERT, UPDATE and DELETE—are not permanent until they are committed. *Explicitly* include COMMIT statements in your script to commit DML changes.

To undo all database changes since the last COMMIT, use ROLLBACK.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

CHAPTER 13

Firebird Data Types

When creating a new column in a Firebird table, the primary attribute that you must define is the data type, which establishes the set of valid data that the column can contain. Only values that can be represented by that data type are allowed.



OTHER REFERENCES

For a complete list of data types supported by Firebird, see *Firebird Reference Guide—Supported Datatypes* (ch. 5 p. 254)

About Firebird data types

Firebird supports most SQL data types. In addition, it supports binary large object (BLOB), a dynamically sizable data type and homogeneous, multi-dimensional arrays of any data type except BLOB or ARRAY.

Besides establishing the set of valid data that a column can contain, the data type defines the operations that can be performed on the data. For example, numbers in INTEGER columns can be manipulated with arithmetic operations, while CHARACTER columns cannot.

The data type also determines how much space each data item occupies on the disk. Choosing an optimum size for the data value is an important consideration when disk space or network capacity is limited, especially if a table is very large.

Supported data types

- INT_64, INTEGER and SMALLINT
- FLOAT and DOUBLE PRECISION
- NUMERIC and DECIMAL
- DATE, TIME, and TIMESTAMP
- CHARACTER, VARYING CHARACTER, and NATIONAL CHARACTER



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

- [BLOB](#)

BLOBs

A BLOB is used to store very large text or binary objects of indeterminate and variable size, such as bitmapped graphics images, vector drawings, sound files, video segments, chapter or book-length documents, or any other kind of multimedia information. The Firebird engine itself is not concerned with the format of BLOB content.

Firebird provides two BLOB sub-types to store data that cannot easily be stored in one of the standard SQL data types. A specialized text BLOB sub-type can be manipulated in several ways as a string. Firebird can also handle externally-defined blob filters, which can be declared as BLOB sub-types, to transform BLOBS from one format to another.

Arrays

Firebird supports arrays of most data types. An *array* is a matrix of individual items composed of any single Firebird data type except BLOB. An array can have from 1 to 16 dimensions. It can be handled as a single entity, or manipulated item-by-item.

Where to specify data types

A data type is assigned to a column in the following situations:

- Creating a table using CREATE TABLE.
- Creating a global column template using CREATE DOMAIN.
- Modifying a global column template using ALTER DOMAIN.
- Adding a new column to a table or altering a column using ALTER TABLE.
- Declaring arguments and local variables in stored procedures and triggers

The syntax for specifying the data type with the DDL statements is provided here for reference.

 For information about data types in stored procedure and trigger variable declarations, please see [Using variables](#) on p. 502 of chapter 25.

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

```
<data type> =
{SMALLINT | INTEGER | FLOAT | DOUBLE PRECISION} [<array_dim>]
| {DATE | TIME | TIMESTAMP} [<array_dim>]
| {DECIMAL | NUMERIC} [(precision [, scale])] [<array_dim>]
| {CHAR | CHARACTER | CHARACTER VARYING | VARCHAR} [(int)]
[<array_dim>] [CHARACTER SET charname]
| {NCHAR | NATIONAL CHARACTER | NATIONAL CHAR}
[VARYING] [(int)] [<array_dim>]
| BLOB [SUB_TYPE {int | subtype_name}] [SEGMENT SIZE int]
[CHARACTER SET charname]
| BLOB [(seglen [, subtype])]
```

For more information on how to create a data type using CREATE TABLE and ALTER TABLE, see chapter 16, [Tables](#) (p. 257).

For more information on using CREATE DOMAIN to define data types, see [Firebird domains](#) on p. 236 of chapter 14.

Fixed-decimal (scaled) data types

Firebird provides two fixed decimal or scaled data types: NUMERIC and DECIMAL. Either scaled type is declared as TYPE(P, S) with P indicating precision (number of significant digits) and S indicating scale (number of decimal places, i.e. digits to the right of the decimal point symbol)

For example, NUMERIC(15, 2) defines a number consisting of up to 15 digits, including two digits to the right of the decimal point.

When you create a domain or column with a NUMERIC or DECIMAL data type, Firebird determines which data type to use for internal storage according to the precision and scale specified and the dialect of the database.

- NUMERIC and DECIMAL data types, when declared with neither precision nor scale are stored as INTEGER.

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

- Defined with precision, with or without scale, they are stored as SMALLINT, INTEGER, DOUBLE PRECISION or 64-bit integer. Storage type depends on both the precision and the dialect of the database. Table 12–53 describes these relationships.

TABLE 13–1 How Firebird stores NUMERIC and DECIMAL data types

Precision	Dialect 1	Dialect 3
1 to 4	SMALLINT for NUMERIC data types INTEGER for DECIMAL data types	SMALLINT
5 to 9	INTEGER	INTEGER
10 to 18	DOUBLE PRECISION	INT64

❖ NUMERIC and DECIMAL data types with precision greater than 10 always produce an error when you create a dialect 2 database. This forces you to examine each instance during a migration.

For a general discussion of migration issues, see chapter 27, [Migrating to Firebird](#) (p. 617)

The following table summarizes how Firebird stores NUMERIC and DECIMAL data types according to precision and scale:

data type specified as...	data type stored as...
NUMERIC	INTEGER
NUMERIC(4)	SMALLINT
NUMERIC(9)	INTEGER
NUMERIC(10)	<ul style="list-style-type: none">DOUBLE PRECISION in dialect1INT64 in dialect 3
NUMERIC(4,2)	SMALLINT



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

data type specified as... data type stored as...

NUMERIC(9,3)	INTEGER
NUMERIC(10,4)	<ul style="list-style-type: none">• DOUBLE PRECISION in dialect1• INT64 in dialect 3
DECIMAL	INTEGER
DECIMAL(4)	INTEGER
DECIMAL(9)	INTEGER
DECIMAL(10)	<ul style="list-style-type: none">• DOUBLE PRECISION in dialect1• INT64 in dialect 3
DECIMAL(4,2)	INTEGER
DECIMAL(9,3)	INTEGER
DECIMAL(10,4)	<ul style="list-style-type: none">• DOUBLE PRECISION in dialect1• INT64 in dialect 3

NUMERIC and DECIMAL with precision and scale

When a NUMERIC or DECIMAL data type declaration includes both precision and scale, values containing a fractional portion can be stored, and you can control the number of fractional digits. Firebird stores such values internally as SMALLINT, INTEGER, or 64-bit integer data, depending on the precision specified. For all SMALLINT and INTEGER data entered, Firebird stores:

- A *scale factor*, a negative number indicating how many decimal places are contained in the number, on base 10. A scale factor of -1 indicates a fractional portion of tenths; a -2 scale factor indicates a fractional portion of hundredths. You do not need to include the sign; it is negative by default.

For example, when you specify NUMERIC(4,2), Firebird stores the number internally as a SMALLINT. If you insert the number 25.253, it is stored as a decimal 25.25, with 4 digits of precision, and a scale of 2.

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

- The number is divided by 10 to the power of *scale* (number/ 10^{scale}) to produce a number without a fractional portion.
- See see [Using exact numeric data types in arithmetic](#) on page 238 for information about arithmetic operations using exact and approximate numerics.

Numeric input and exponents

Any numeric string in **dsql** or **isql** that can be stored as a DECIMAL(18,S) is evaluated exactly, without the loss of precision that might result from intermediate storage as a DOUBLE. A numeric string is recognized by the **dsql** parser as a floating-point value only if it contains an "e" or "E" followed by an exponent, which may be zero. For example, **dsql** recognizes 4.21 as a scaled exact integer, and passes it to the engine in that form. On the other hand, **dsql** recognizes 4.21E0 as a floating-point value.

Integer data types: SMALLINT, INTEGER and INT64

Integers are whole numbers. Firebird supports two named integer data types: SMALLINT and INTEGER and NUMERIC(18,0), a 64-bit integer.

- SMALLINT is a signed short integer with a range from -32,768 to 32,767.
- INTEGER is a signed long integer with a range from -2,147,483,648 to 2,147,483,647.
- NUMERIC(18,0) is a signed 64-bit integer with a range from -10^{63} to $10^{63}-1$.
- From Firebird 1.5 forward, INT64 is available as a discrete type

All are *exact numerics, signed*. Firebird does not support an unsigned integer type.

The next two statements create domains with the SMALLINT and INTEGER data types:

```
CREATE DOMAIN EMPNO  
    AS SMALLINT;  
  
CREATE DOMAIN CUSTNO  
    AS INTEGER  
    CHECK (VALUE > 99999);
```

You can perform the following operations on the integer data types:

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

- Comparisons using the standard relational operators ($=$, $<$, $>$, \geq , \leq). Other operators such as CONTAINING, STARTING WITH, and LIKE perform string comparisons on numeric values.
- Arithmetic operations. The standard arithmetic operators determine the sum, difference, product, or quotient of two or more integers.
- Conversions. When performing arithmetic operations that involve mixed data types, Firebird automatically converts between INTEGER, FLOAT, and CHAR data types. For operations that involve comparisons of numeric data with other data types, Firebird first converts the data to a numeric type, then performs the arithmetic operation or comparison.
- Sorts. By default, a query retrieves rows in the exact order that it finds them in the table, which is likely to be unordered. You can sort rows on integer columns using the ORDER BY clause of a SELECT statement in descending or ascending order.

Fixed-decimal data types: NUMERIC and DECIMAL

Firebird supports two SQL data types, NUMERIC and DECIMAL, for handling numeric data with a fixed decimal point, such as monetary values. You can specify optional precision and scale factors for both data types. These data types are also referred to as *exact numerics*.

- *Precision* is the total number or maximum number of digits, both significant and fractional, that can appear in a column of these data types. The allowable range for precision is from 1 to a maximum of 18.
- *Scale* is the number of digits to the right of the decimal point that comprise the fractional portion of the number. The allowable range for scale is from zero to *precision*; in other words, scale must be less than or equal to precision.

The syntax for NUMERIC and DECIMAL is as follows:

```
NUMERIC[ (precision [, scale]) ]  
DECIMAL[ (precision [, scale]) ]
```

You can specify NUMERIC and DECIMAL data types without precision or scale, with precision only, or with both precision and scale.

NUMERIC data type

NUMERIC(x,y)

**Symbols** **Numerics** A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

In the syntax above, Firebird stores exactly x digits. Of that number, exactly y digits are to the right of the decimal point. For example,

`NUMERIC(5,2)`

declares that a column of this type always holds numbers with exactly 5 digits, with exactly two digits to the right of the decimal point: `sss.aa`.

DECIMAL data type

`DECIMAL(x,y)`

In the syntax above, Firebird stores at least x digits. Of that number, exactly y digits are to the right of the decimal point. For example,

`DECIMAL(5,2)`

declares that a column of this type must be capable of holding at least five but possibly more digits and exactly two digits to the right of the decimal point: `sss.aa`.

Using exact numeric data types in arithmetic

In **SQL dialect 1**, when you divide two integers or two `DECIMAL(9,2)` values, the quotient is a floating-point value of type `DOUBLE PRECISION`.

In **SQL dialect 3**, the quotient of two exact numeric values—`SMALLINT`, `INTEGER`, `NUMERIC(n,m)` or `DECIMAL(n,m)`—is an exact numeric, with scale factor equal to the sum of the scales of the dividend and divisor.

Illustrations

The quotient of two `INTEGER`s is an `INTEGER` because a `SMALLINT`, `INTEGER` or `INT_64` has a scale of 0. The quotient of a `DECIMAL(9,2)` and a `DECIMAL(12,3)` is a `DECIMAL(18,5)`.

In dialect 1, the fraction $1/3$ is $0.333333333333e0$ because the quotient is `DOUBLE PRECISION`

In dialect 3, $1/3$ results in 0 because the zero scale of the two operands.

- ❖ When an application does something that causes a `CHECK` condition to be checked, or a stored procedure to be executed, or a trigger to fire, the processing that takes place is based on the

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

dialect under which the check, stored procedure, or trigger was *defined*, not the dialect in effect when the application causes the check, stored procedure, or trigger to be *executed*.

For example, suppose that a database is migrated from InterBase 5 and thus has dialect 1; that MYCOL1 and MYCOL2 are SQL INTEGERS; and that a table definition includes the following:

```
CHECK (MYCOL1 / MYCOL2 > 0.5)
```

which was defined using client dialect 1.

Now suppose that a dialect 3 client tries to insert a row in which MYCOL1 is 3 and MYCOL2 is 5: because the CHECK was defined in dialect 1, the quotient will be 0.600000000000e0, and the row will pass the check condition, even though in the current client's dialect, 3, the quotient would have been the integer 0, and the row would have failed the check, causing the insert to fail.

Operations using exact numerics

All NUMERIC and DECIMAL data types are stored as exact numerics: 16, 32, or 64 bits, depending on the precision. NUMERIC and DECIMAL data types with precision greater than 9 are referred to as *large exact numerics*.

- If one operand is an approximate numeric, the result of any dyadic operation (addition, subtraction, multiplication, division) is DOUBLE PRECISION.
- Any value that can be stored in a DECIMAL(18,S) can also be specified as the default value for a column or a domain.

Addition and subtraction

If both operands are exact numeric, adding or subtracting the operands produces an exact numeric with a precision of 18 and a scale equal to the larger of the two. For example:

```
CREATE TABLE t1 (n1 NUMERIC(16,2), n2 NUMERIC(16,3));
INSERT INTO t1 VALUES (12.12, 123.123);
COMMIT;
```

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

The following query returns the integer 135.243. The largest scale of the two operands is 3; therefore, the scale of the sum is 3.

```
SELECT n1 + n2 FROM t1;
```

Similarly, the following query returns the integer -111.003:

```
SELECT n1 - n2 FROM t1;
```

If either of the operands is approximate numeric (FLOAT, REAL, or DOUBLE PRECISION), the result is DOUBLE PRECISION.

Multiplication

If both operands are exact numeric, multiplying the operands produces an exact numeric with a precision of 18 and a scale equal to the sum of the scales of the operands. For example:

```
CREATE TABLE t1 (n1 NUMERIC(16,2), n2 NUMERIC(16,3));
INSERT INTO t1 VALUES (12.12, 123.123);
COMMIT;
```

The following query returns the integer 1492.25076 because n1 has a scale of 2 and n2 has a scale of 3. The sum of the scales is 5.

```
SELECT n1*n2 FROM t1
```

If one of the operands is approximate numeric (FLOAT, REAL, or DOUBLE PRECISION), the result is DOUBLE PRECISION.

Division

If both operands are exact numeric, dividing the operands produces an exact numeric with a precision of 18 and a scale equal to the sum of the scales of the operands. If at least one operand of a division operator has an approximate numeric type (FLOAT, REAL, or DOUBLE PRECISION), the result is DOUBLE PRECISION.

For example, in the following table defined in Firebird 1, division operations produce a variety of results:

```
CREATE TABLE t1 (i1 INTEGER, i2 INTEGER, n1 NUMERIC(16,2),
n2 NUMERIC(16,2));
INSERT INTO t1 VALUES (1, 3, 1.00, 3.00);
COMMIT;
```

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

The following query returns the integer 0 because each operand has a scale of 0, so the sum of the scales is 0:

```
SELECT i1/i2 FROM t1
```

The following query returns the NUMERIC(18,2) value 0.33, since the sum of the scales 0 (operand 1) and 2 (operand 2) is 2.

```
SELECT i1/n2 from t1
```

The following query returns the NUMERIC(18,4) value 0.3333, since the sum of the two operand scales is 4.

```
SELECT n1/n2 FROM t1
```

In InterBase 5 and older, any of the above division operations return the DOUBLE PRECISION value 0.3333333333333333.

To obtain an InterBase 5 result when using Firebird 1, alter your query to cast at least one of the operands into an approximate type. For example,

```
SELECT i1/cast(i2 as double precision) from t1;
```

Specifying numeric data types in embedded applications

DSQL applications such as `isql` can correct for the scale factor for SMALLINT and INTEGER data types by examining the `XSQLVAR sqlscale` field and dividing to produce the correct value.

Both SQL and DSQL applications handle NUMERIC and DECIMAL types stored as 64-bit integers without a problem.

- ❖ Embedded applications cannot use or recognize small precision NUMERIC or DECIMAL data types with fractional portions when they are stored as SMALLINT or INTEGER types. To avoid this problem, any NUMERIC and DECIMAL data types that are to be accessed from embedded applications should be created with a precision of 10 or more, which forces them to be stored as 64-bit integer types. Again, remember to specify a scale if you want to control the precision and scale.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Floating-point data types

Firebird provides two floating-point data types, FLOAT and DOUBLE PRECISION, differing only in size:

- FLOAT specifies a single-precision, 32-bit data type. Its precision is approximately 7 decimal digits.
- DOUBLE PRECISION specifies a double-precision, 64-bit data type. Its precision is approximately 15 decimal digits.

The precision of FLOAT and DOUBLE PRECISION is fixed by their size. The scale is not fixed and it cannot be controlled. By the nature of "float" types, placement of the decimal point can vary. For example, it is valid to store, in the same column, one value as 25.33333, and another as 25.333.

If the value stored is outside the precision limits of the floating-point type, it is stored only approximately, its least-significant digits being treated as zeros. Type FLOAT, for example, limits precision to 7 digits. A 10-digit number 25.33333312 inserted into a FLOAT column is stored as 25.33333.

The next statement creates a column, PERCENT_CHANGE, using a DOUBLE PRECISION type:

```
CREATE TABLE SALARY_HISTORY
(
    .
    .
    .
    PERCENT_CHANGE DOUBLE PRECISION
    DEFAULT 0
    NOT NULL
    CHECK (PERCENT_CHANGE BETWEEN -50 AND 50),
    .
    .
    .
);
```

The following operations can be performed on FLOAT and DOUBLE PRECISION data types:

- **Comparisons** using the standard relational operators ($=, <, >, >=, <=$). Other operators such as CONTAINING, STARTING WITH, and LIKE perform string comparisons on the integer portion of floating data.
- **Arithmetic operations** The standard arithmetic operators determine the sum, difference, product, or quotient of two or more integers.
- **Conversions** When performing arithmetic operations that involve mixed data types, Firebird automatically converts between INTEGER, FLOAT, and CHAR data types. For operations that involve

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

comparisons of numeric data with other data types, such as CHARACTER and INTEGER, Firebird first converts the data to a numeric type, then compares them numerically.

- **Sorts** By default, a query retrieves rows in the exact order that it finds them in the table, which is likely to be unordered. Sort rows using the ORDER BY clause on a FLOAT or DOUBLE PRECISION column of a SELECT statement, in descending or ascending order.
- **Use floating-point numbers** when you expect the placement of the decimal point to vary, and for storing numeric data values having a very wide range, such as those for scientific calculations.

Example using numerical data types

The following CREATE TABLE statement provides an example of how the different numeric types can be used: an INTEGER for the total number of orders, a fixed DECIMAL for the dollar value of total sales, and a FLOAT for a discount rate applied to the sale.

```
CREATE TABLE SALES
( . . .
  QTY_ORDERED INTEGER
    DEFAULT 1
    CHECK (QTY_ORDERED >= 1),
  TOTAL_VALUE DECIMAL (9,2)
    CHECK (TOTAL_VALUE >= 0),
  DISCOUNT FLOAT
    DEFAULT 0
    CHECK (DISCOUNT >= 0 AND DISCOUNT <= 1));
```

The DATE, TIME, and TIMESTAMP data types

Firebird supports DATE, TIME, and TIMESTAMP data types.

- DATE stores a date as a 32-bit longword. Valid dates are from January 1, 0001 to December 31, 9999
- TIME stores time as a 32-bit longword. Valid times are from 00:00 to 23:59:59.9999.
- TIMESTAMP is stored as two 32-bit longwords and is a combination of DATE and TIME.

**Symbols** [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

data type	Definition	Size	Range of values
DATE	ISC_DATE	32 bits, signed	1 Jan 0001 to 31 Dec 9999
TIME	ISC_TIME	32 bits, unsigned	00:00:00.0000 to 23:59:59.9999
TIMESTAMP	ISC_TIMESTAMP	64 bits	1 Jan 0001 at 00:00:00.0000 to 31 Dec 9999 at 23:59:59.9999

- The dialect 3 TIMESTAMP data type is a composite data type consisting of a date portion and a time portion, equivalent to the DATE type in dialect 1.
- The dialect 3 DATE and TIME types have no equivalents in dialect 1.

Date and time value context variables

Date and time context variables CURRENT_DATE, CURRENT_TIME, and CURRENT_TIMESTAMP return date and time values capturing the moment of execution of the containing SQL statement, using the server's clock and time zone. Table 12–54 shows data types and meanings of these variables.

TABLE 13–2 Date and time context variables

Function	data type	Meaning
CURRENT_TIMESTAMP	TIMESTAMP	Current date and time to the nearest 10,000 th of a second.
CURRENT_DATE	DATE	Current date.
CURRENT_TIME	TIME	Current time, expressed as 10,000 th s of a second since midnight.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Date literals

Date literals are strings of specific formats, used with single quotation marks, that Firebird server recognizes as date or date-and-time constants for EXTRACT, INSERT, and UPDATE operations and in the WHERE clause of a SELECT statement.

Specifically, date literals are used when supplying date constants to

- SELECT, UPDATE and DELETE statements, in the search condition of a WHERE clause
- INSERT and UPDATE statements, to enter date and time constants
- the second argument of the EXTRACT function.

Recognized formats for date literals

TABLE 13-3 Symbolic segments of date literals

Segment	Representing
CC	Century - first two digits of a year segment, e.g. '20' for the twenty-first century
YY	Year in century. Firebird always stores the full year value if the year is entered without the 'CC' segment, using a "sliding window" algorithm to determine which century to store. For details, see Firebird's "sliding century window" .
MM	Month, evaluating to an integer in the range 1 to 12. In some formats, two digits are required.
MMM	Month, one of [JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC] English month names fully spelt out (correctly) are also valid.



CHAPTER 13 Firebird Data Types } The DATE, TIME, and TIMESTAMP data types

Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

TABLE 13–3 Symbolic segments of date literals

Segment	Representing
DD	Day of the month, evaluating to an integer in the range 1 to 31. In some formats, two digits are required. An invalid day-of-month number for the given month will cause an error.
HH	Hours, evaluating to an integer in the range 00 to 23. Two digits are required when storing a time portion.
Min	Minutes, evaluating to an integer in the range 00 to 59. Two digits are required when storing a time portion.
SS	Whole seconds, evaluating to an integer in the range 00 to 59. Two digits are required when storing a time portion.
nnnn	Ten-thousandths of a second in the range zero to 9999. Optional for time portions, defaults to 0000. If used, four digits are required.

The recognized formats are:

TABLE 13–4

Format	Dialect 1 DATE	Dialect 3 DATE	Dialect 3 TIMESTAMP
'CCYY-MM-DD' or 'YY-MM-DD'	Stores date and a time portion of 00:00:00	Stores date only	Stores date and a time portion of 00:00:00
'MM/DD/CCYY' or 'MM/DD/YY'	as above	as above	as above
'DD.MM.CCYY' or 'DD.MM.YY'	as above	as above	as above



CHAPTER 13 Firebird Data Types } The DATE, TIME, and TIMESTAMP data types

Symbols **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

TABLE 13–4

Format	Dialect 1 DATE	Dialect 3 DATE	Dialect 3 TIMESTAMP
'DD-MMM-CCYY' or 'DD-MMM-YY' plus variants using blanks or commas as separators, or using no separators. English month names fully spelt out are also valid.	as above	as above	as above
CCYY-MM-DD HH:Min:SS.nnnn' or 'YY-MM-DD HH:Min:SS.nnnn'	Stores date and time. ".nnnn" segment is optional	Stores date only: may need to be CAST as date. Time portion is not stored.	Stores date and time. ".nnnn" segment is optional
'MM/DD/CCYY HH:Min:SS.nnnn' or 'MM/DD/YY HH:Min:SS.nnnn'	as above	as above	as above
'DD.MM.CCYY HH:Min:SS.nnnn' or 'DD.MM.YY HH:Min:SS.nnnn'	as above	as above	as above
'DD-MMM-CCYY HH:Min:SS.nnnn' or 'DD-MMM-YY HH:Min:SS.nnnn'	as above	as above	as above

The TIMESTAMP data type (or the dialect 1 DATE data type) accepts both date and time portions of a date literal. The dialect 3 DATE data type accepts only the date portion. The TIME data type accepts only the time portion. Firebird recognizes the following string formats as date literals. Optional components are shown in brackets:

Firebird interprets a string such as '6/29/2000 01:06:00' as a timestamp in the following contexts:
When the situation calls for an implicit cast from string to date:

```
INSERT INTO MyTable (DateField) VALUES ('6/29/2000')
```



CHAPTER 13 Firebird Data Types } The DATE, TIME, and TIMESTAMP data types

Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

or

```
SELECT ... WHERE DateField > '6/29/2000'
```

When an explicit cast changes a string to a date:

```
SELECT CAST('6/29/2000' AS DATE) ...
```

Separators

- The forward slash separator (“/”) is not available for any format other than the US MM/DD/[CC]YY. Use of this separator will cause wrong dates or range errors if you try to apply it to any non-US date system.
- Using “.” (fullstop or period) in either of the first two separation points in formats where **MM** is numeric causes the date to be parsed in international order: Day, Month, Year. A format **CCYY.MM.DD** will cause wrong dates or range errors if you try to apply it to the US date system.

Whitespace

- Any whitespace (spaces or tabs) can appear between segments but there must be at least one space separating a date portion from a time portion.

Examples

All of the following represent the 30th day of the 6th month (June) of the year 2000:

```
'30 jun 2000''06/30/2000''Jun 30 00'  
'30-jun-2000''30.6.2000''30 JUN 00'  
'30,jun,2000''6-30-00''2000,30,06'  
'30.jun.2000''2000 June 30''2000-jun-30'  
'30jun2000'      'June 30, 2000' '6,30,2000'
```



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Predefined date literals

Firebird supports the following special date literals: 'NOW', 'TODAY', 'YESTERDAY', and 'TOMORROW'. Table 12-57 shows formats and meanings of these date literals.

TABLE 13-5 Predefined date literals

Date Literal	Format	Meaning
'NOW'	TIMESTAMP	Current date and time to the nearest 10,000 th of a second since midnight, as of the start of the operation. Equivalent to the context variable CURRENT_TIMESTAMP.
'TODAY'	DATE	Current date, as of the start of the operation. Equivalent to the context variable CURRENT_DATE.
'TOMORROW'	DATE	Current date as of the start of the operation, plus one day.
'YESTERDAY'	DATE	Current date as of the start of the operation, minus one day.

'NOW' can be stored in DATE, TIME, and TIMESTAMP fields. Only the date portion is stored in a DATE field, and only the time portion is stored in a TIME field. Both date and time portions are stored in a TIMESTAMP field.

'TODAY', 'TOMORROW', and 'YESTERDAY' can be stored in DATE and TIMESTAMP fields, but not in TIME fields. In each case, the date value is stored. In a TIMESTAMP field, the time portion is set to midnight (00:00:00.0000).

The TIME data type

Firebird can store a TIME data type representing the time portion alone of any instant in a 24-hour period. The literal format for TIME constants is 'HH:Min:SS.nnnn'. As with timestamps, the fractional part of the seconds segment ("nnnn") is optional. Any time between midnight ('00:00:00') and '23:59:59.9999' is valid.

- ★ You can capture the current server time as a TIME value using the context variable CURRENT_TIME.

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

- ❖ Don't make the mistake of thinking that the TIME data type records an *interval of time*. Use a column of an integer data type for storing intervals of time, which can be calculated arithmetically in expressions involving DATE, TIMESTAMP or TIME types.

Firebird's "sliding century window"

Firebird always stores the full year value in a DATE or TIMESTAMP column, never the two-digit abbreviated value. When a client application enters a two-digit year value, Firebird uses the "sliding window" algorithm, described below, to make an inference about the century and stores the full date value including the century. When you retrieve the data, Firebird returns the full year value including the century information. Client applications are responsible for displaying the year as two or four digits.

Firebird uses the following *sliding window* algorithm to infer a century:

- Compare the two-digit year number entered to the current year modulo 100
- If the absolute difference is greater than 50, then infer that the century of the number entered is 20, otherwise it is 19.

A two-digit year value is interpreted to be the nearest year to the current year, in a 50-year range. For example, if the current year were 1996, two-digit year values would be interpreted thus:

TABLE 13-6 Example: how two-digit years are interpreted relative to current year 1996

This two-digit year	Becomes this four-digit year	This two-digit year	Becomes this four-digit year
96	1996	44	2044
97	1997	45	2045
00	2000	46	1946
01	2001	47	1947



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Operations using date and time values

Add an integer n to, or subtract an integer n from a DATE or TIMESTAMP value to change the value by n days.

Adding an integer n to, or subtract an integer n from a TIME value to change the value by n seconds.

Change the time part of a TIMESTAMP value by an integral number of hours, minute, or seconds by adding a fraction of a day, as a constant or as an expression:

- to add n seconds, add $n/86400.0$
- to add n minutes, add $n/3600.0$
- to add n hours, add $n/60.0$

❖ In dialect 3, if n is less than the divisor, one of the operands must be floating point, to avoid a result of zero arising from the division of one integer operand by another.

★ There is no built-in way to add months, years, or quarters to a TIMESTAMP value in a single operation.

Table 12–59 shows the result of adding and subtracting DATE, TIME, TIMESTAMP, and exact numeric (INTEGER, DECIMAL, or NUMERIC) values:

TABLE 13–7 Adding and subtracting date/time data types

Operand 1	Operator	Operand 2	Result
DATE	+	DATE	Error
DATE	+	TIME	TIMESTAMP (concatenation)
DATE	+	TIMESTAMP	Error
DATE	+	Numeric value	DATE + number of days: fractional part ignored
TIME	+	DATE	TIMESTAMP (concatenation)

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**TABLE 13-7 Adding and subtracting date/time data types (*continued*)

Operand 1	Operator	Operand 2	Result
TIME	+	TIME	Error
TIME	+	TIMESTAMP	Error
TIME	+	Numeric value	TIME + number of seconds: 24-hour modulo arithmetic
TIMESTAMP	+	DATE	Error
TIMESTAMP	+	TIME	Error
TIMESTAMP	+	TIMESTAMP	Error
TIMESTAMP	+	Numeric value	TIMESTAMP: DATE + number of days; TIME + fraction of day converted to seconds
DATE	-	DATE	DECIMAL(9,0) representing number of days
DATE	-	TIME	Error
DATE	-	TIMESTAMP	Error
DATE	-	Numeric value	DATE – number of days: fractional part ignored
TIME	-	DATE	Error
TIME	-	TIME	DECIMAL(9,4) representing number of seconds
TIME	-	TIMESTAMP	Error
TIME	-	Numeric value	TIME – number of seconds: 24-hour modulo arithmetic
TIMESTAMP	-	DATE	Error
TIMESTAMP	-	TIME	Error

**Symbols** [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)TABLE 13-7 Adding and subtracting date/time data types (*continued*)

Operand 1	Operator	Operand 2	Result
TIMESTAMP	-	TIMESTAMP	DECIMAL(18,9) representing days and fraction of day
TIMESTAMP	-	Numeric value	TIMESTAMP: DATE – number of days; TIME – fraction of day converted to seconds

The following statement creates TIMESTAMP columns in the SALES table:

```
CREATE TABLE SALES
  ( . . .
    ORDER_DATE TIMESTAMP
      DEFAULT 'now'
      NOT NULL,
    SHIP_DATE TIMESTAMP
      CHECK (SHIP_DATE >= ORDER_DATE OR SHIP_DATE IS NULL),
  . . . );
```

In the previous example, 'NOW' returns the system date and time.

Converting to the DATE, TIME, and TIMESTAMP data types

Most languages do not support the DATE, TIME, and TIMESTAMP data types. Instead, they express them as strings or structures. These data types require conversion to and from Firebird when entered or manipulated in a host-language program. For example, you could convert to the DATE data type in one of the following ways:

- Create a string in a format that Firebird understands (for example, 1-JAN-1999). When you insert the date into a DATE column, Firebird automatically converts the text into the internal DATE format.
- ❖ The string conversion does not work in the other direction. To read a date in a Firebird format and convert it to a C date variable, you must call *isc_decode_date()*.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- Use the call interface routines provided by Firebird to do the conversion. *isc_decode_date()* converts from the Firebird internal DATE format to the C time structure. *isc_encode_date()* converts from the C time structure to the internal Firebird DATE format.

Casting between time and date types

The built-in CAST() function in SELECT statements can be used in SELECT statements to translate between date and time datatypes and character-based datatypes. The following conversions are valid:

- **From date types to CHAR(n)**

When converting DATE, TIME, or TIMESTAMP types into a CHAR type, you need to prepare a CHAR column or variable of at least 24 characters.

Alternatively, cast a TIMESTAMP to a DATE and then cast the DATE to a CHAR of a suitable shorter length.

For example:

```
SELECT CAST (CAST (timestamp_col AS DATE) AS CHAR(10)) FROM table1;
```

- **From CHAR(n) to DATE, TIME, or TIMESTAMP**

- You cannot cast a date or time datatype to or from BLOB, SMALLINT, INTEGER, FLOAT, DOUBLE PRECISION, NUMERIC, or DECIMAL datatypes.

The CAST() function

CAST() can be applied in WHERE clause criteria to compare differing data types. The syntax for CAST() is:

```
CAST (<value> AS <datatype>)
```

In the following WHERE clause, the CAST() expression converts a CHAR datatype, INTERVIEW_DATE, to a DATE type for comparing with a DATE data type, HIRE_DATE:

```
... WHERE HIRE_DATE = CAST(INTERVIEW_DATE AS DATE);
```

Next, CAST() converts a DATE type into a CHAR type:

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

```
... WHERE CAST(HIRE_DATE AS CHAR(10)) = INTERVIEW_DATE;
```

CAST() also can be used to compare columns with different datatypes in the same table, or between tables.

Valid CASTs

Casting from other data types to date and time types

The following table shows the SQL datatypes from which the DATE, TIME, and TIMESTAMP datatypes can be cast:

TABLE 13-8 Casting other data types as date/time types

SMALLINT, INTEGER, FLOAT, DOUBLE PRECISION, NUMERIC, DECIMAL and BLOB types cannot be cast as any date/time type

Data Type	AS TIMESTAMP	AS DATE	AS TIME
VARCHAR(n)	Yes, if the string is a valid Firebird date literal which includes HH:Mins:SS.nnnn	Yes, if the string is a valid Firebird date literal	Yes, if the string has the format HH:Mins:SS.nnnn
TIMESTAMP	Yes	Yes: generates date portion of the timestamp	Yes: generates the time portion of the timestamp
DATE	Yes: generates date plus a time portion as 00:00:00.0000	Yes	Causes an error
TIME	Yes: generates the Firebird base date of 17.11.1858 in the date portion	Causes an error	Yes



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Casting from date and time datatypes to other SQL datatypes

Date and time types can not be cast as any of the following: SMALLINT, INTEGER, FLOAT, DOUBLE PRECISION, NUMERIC, DECIMAL and BLOB types

The following table shows the data types into which the TIMESTAMP, DATE AND TIME types can be cast:

TABLE 13-9 Casting date/time types as other types

Data Type	FROM TIMESTAMP	FROM DATE	FROM TIME
AS VARCHAR(n) CHAR(n)	Yes, if the string is 24 or more characters long. The resulting string is in the format CCYY-MM-DD HH:Mins:SS.nnnn	Yes, if the string is 10 or more characters long. The resulting string is in the format CCYY-MM-DD	Yes, if the string is 10 or more characters long. The resulting string is in the format HH:Mins:SS.nnnn
AS TIMESTAMP	Yes	Yes: generates date plus a time portion as 00:00:00.0000	Yes: generates the Firebird base date of 17.11.1858 in the date portion
AS DATE	Yes: generates date portion of the timestamp	Yes	Causes an error
AS TIME	Yes: generates the time portion of the timestamp	Causes an error	Yes



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Character data types

Firebird supports four character string data types:

- 1 A fixed-length character data type, called CHAR(*n*) or CHARACTER(*n*), where *n* is the *exact* number of characters stored.
- 2 A variable-length character type, called VARCHAR(*n*) or CHARACTER VARYING(*n*), where *n* is the *maximum* number of characters in the string.
- 3 An NCHAR(*n*) or NATIONAL CHARACTER(*n*) or NATIONAL CHAR(*n*) data type, which is a fixed-length character string of *n* characters that uses the ISO8859_1 character set.
- 4 An NCHAR VARYING(*n*) or NATIONAL CHARACTER VARYING(*n*) or NATIONAL CHAR VARYING(*n*) data type, which is a variable-length national character string up to a maximum of *n* characters.

Specifying a character set

When you define the data type for a column, you can specify a character set for the column with the CHARACTER SET argument. This setting overrides the database default character set that is assigned when the database is created.

You can also change the default character set with SET NAMES in command-line isql.

The character set determines:

- the characters that can be used in CHAR, VARCHAR, and BLOB SUB_TYPE 1 (text) columns.
- the number of bytes allocated to each character
- the default collation order to be used in sorting on the column.

★ Collation order does not apply to BLOB data.

For example, the following statement creates a column that uses the ISO8859_1 character set, which is used to support several European languages:

```
CREATE TABLE EMPLOYEE  
  (FIRST_NAME VARCHAR(10) CHARACTER SET ISO8859_1,  
  . . .);
```

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

For a list of the international character sets and collation orders that Firebird supports, see chapter 15, [Character Sets and Collation Orders](#) (p. 246).

Characters vs. bytes

The number of bytes that the system uses to store a single character can vary according to the character set. Firebird limits a character column to 32,767 bytes and an index width to 253 bytes. Some character sets require two or three bytes per character. Accordingly, the maximum number of characters allowed in n depends upon the character set used.

- In a character column defined to use a single-byte characters, the column can be up to 32,767 characters without encountering a validity error in the DDL statement.

In a character column defined to use multi-byte characters, one character does *not* equal one byte.

★ To determine the maximum number of characters allowed in the data definition statement of any multi-byte column, look up the number of bytes per character in XXXXX. Then divide 32,767 (the internal byte storage limit for any character data type) by the number of bytes for each character. Two-byte character sets have a character limit of 16,383 per field, and a three-byte character set has a limit of 10,922 characters per field.

In the following example, the user specifies a CHAR data type using the UNICODE_FSS character set:

```
CHAR (10922) CHARACTER SET UNICODE_FSS; /* succeeds */  
CHAR (10923) CHARACTER SET UNICODE_FSS; /* fails */
```

This character set has a maximum size of 3 bytes for a single character. Because each character requires 3 bytes of internal storage, the maximum number of characters allowed without encountering an error is 10,922 (32,767 divided by 3 is approximately 10,922).

Using CHARACTER SET NONE

If a default character set was not specified when the database was created, the character set defaults to NONE. Using CHARACTER SET NONE means that there is no character set assumption for columns; character data are stored and retrieved just as you originally entered them.

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z****Transliteration errors**

You can load any character set into a column defined with NONE, but you cannot load that same data into another column that has been defined with a different character set. No transliteration will be performed between the source and destination character sets, so in most cases, errors will occur during the attempted assignment.

For example:

```
CREATE TABLE MYDATA (PART_NUMBER CHARACTER(30) CHARACTER SET NONE);
SET NAMES LATIN1;
INSERT INTO MYDATA (PART_NUMBER) VALUES('à');
SET NAMES DOS437;
SELECT * FROM MYDATA;
```

The data ("à") is returned just as it was entered, without the à being transliterated from the input character (LATIN1) to the output character (DOS437). If the column had been set to anything other than NONE, the transliteration error would have occurred.

About collation order

Collation order determines the precedence of the characters in the character set for sorting and searching. Because each character set has its own subset of possible collation orders, the character set that you choose when you define the data type limits your choice of collation orders. You specify the collation order for a column when you create the table.

For a list of the international character sets and collation orders that Firebird supports, see chapter 15, [Character Sets and Collation Orders](#) (p. 246).

Fixed-length character data

Firebird supports two fixed-length string data types: CHAR(n), or alternately CHARACTER (n), and NCHAR(n), or alternately NATIONAL CHAR(n).

CHAR(n) or CHARACTER(n)

The CHAR(n) or CHARACTER(n) data type contains character strings. The number of characters n is fixed.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- To determine the maximum number of characters allowed for the character set that you have specified, see chapter 15, [Character Sets and Collation Orders](#) (p. 246)

When the string to be stored or read contains less than *n* characters, Firebird pads to the right with blanks to make up the difference. If a string is larger than *n*, the value is truncated. If you do not supply *n*, it will default to 1, so CHAR is the same as CHAR(1). The next statement illustrates this:

```
CREATE TABLE SALES
  (
    . . .
    PAID CHAR
      DEFAULT 'n'
      CHECK (PAID IN ('y', 'n')), . . .);
```

Trailing blanks Firebird compresses trailing blanks when it stores fixed-length strings, so data with trailing blanks uses the same amount of space as an equivalent variable-length string. When the data is read, Firebird reinserts the blanks. This saves disk space when the length of the data items varies widely.

NCHAR(*n*) or NATIONAL CHAR(*n*)

NCHAR(*n*) is exactly the same as CHARACTER(*n*), except that it uses the ISO8859_1 character set by definition. Using NCHAR(*n*) is a shortcut for using the CHARACTER SET clause to specify the ISO8859_1 character set for a column.

The next two CREATE TABLE examples are equivalent:

```
CREATE TABLE EMPLOYEE
  (
    . . .
    FIRST_NAME NCHAR(10),
    LAST_NAME NCHAR(15), . . .);

CREATE TABLE EMPLOYEE
  (
    . . .
    FIRST_NAME CHAR(10) CHARACTER SET 'ISO8859_1',
    LAST_NAME CHAR(15) CHARACTER SET 'ISO8859_1', . . .);
```



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Variable-length character data

Firebird supports two variable-length string data types: VARCHAR(*n*), or alternately CHAR(*n*) VARYING, and NCHAR(*n*), or alternately NATIONAL CHAR(*n*) VARYING.

VARCHAR(*n*)

VARCHAR(*n*)—also called CHAR VARYING(*n*), or CHARACTER VARYING(*n*)—allows you to store the exact number of characters that is contained in your data, up to a maximum of *n*. You must supply *n*; there is no default to 1. If the length of the data within a column varies widely, and you do not want to pad your character strings with blanks, use the VARCHAR(*n*) or CHARACTER VARYING(*n*) data type.

Firebird outputs data from variable-length character data as fixed-length strings padded with blanks (spaces) to the full length defined.

The main advantages of using the VARCHAR(*n*) data type are that it saves disk space, and since more rows fit on a disk page, the database server can search the table with fewer disk I/O operations.

- ❖ The disadvantage is that selects and updates can be slower than necessary, if very large varchars are defined to store mostly short strings.

The next statement illustrates the VARCHAR(*n*) data type:

```
CREATE TABLE SALES
(
    ...
    ORDER_STATUS VARCHAR(7)
        DEFAULT 'new'
        NOT NULL
        CHECK (ORDER_STATUS IN ('new', 'open',
            'shipped', 'waiting')), ...
)
```

NCHAR VARYING(*n*)

NCHAR VARYING(*n*)—also called NATIONAL CHARACTER VARYING (*n*) or NATIONAL CHAR VARYING(*n*)—is exactly the same as VARCHAR(*n*), except that the ISO8859_1 character set is used. Using NCHAR VARYING(*n*) is a shortcut for using the CHARACTER SET clause of CREATE TABLE, CREATE DOMAIN, or ALTER TABLE to specify the ISO8859_1 character set.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Defining BLOB data types

Firebird supports a dynamically sizable data type (BLOB) to store binary data or large text objects that cannot easily be stored in one of the standard SQL data types. A BLOB is used to store very large data objects of indeterminate and variable size, such as bitmapped graphics images, vector drawings, sound files, video segments, chapter or book-length documents, or any other kind of multimedia information. Because a blob can hold different kinds of information, it requires special processing for reading and writing.

Use BLOB types when you have file-based data that you prefer to store within your database, rather than as external files linked through pointers stored in the database. Stored in a database, these data gain the benefits of protection from external interference, transaction control, maintenance by database utilities and contextual access by way of SQL statements.

BLOB columns

You define BLOB columns in database tables just as you do non-BLOB columns. For example, the following statement creates a table with a BLOB column of SUB_TYPE 1 (plain text):

```
CREATE TABLE PROJECT
  (PROJ_ID PROJNO NOT NULL,
   PROJ_NAME VARCHAR(20) NOT NULL UNIQUE,
   PROJ_DESC BLOB SUB_TYPE 1,
   TEAM_LEADER EMPNO,
   PRODUCT PRODTYPE,
   . . . );
```

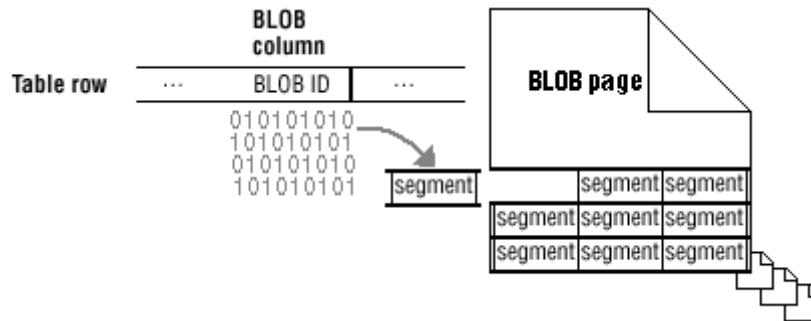
BLOB data are not stored in the structure of their parent tables. They are stored as *segments* in one or more BLOB database pages, linked from the parent table through a BLOB ID. A BLOB ID is a unique hexadecimal pair that references BLOB data. BLOB segments are discrete units of BLOB data that are packaged for reading, writing and transmission across the network one segment at a time.

The following diagram shows the relationship between a BLOB column containing a BLOB ID and the BLOB data referenced by the BLOB ID:



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

FIGURE 10 Blob relationships



BLOB segment length

When a BLOB column is defined in a table, the BLOB definition can specify the expected size of BLOB segments that are written to the column. Actually, for SELECT, INSERT, and UPDATE operations, BLOB segments can be of varying length. For example, during insertion, a BLOB might be read in as three segments, the first segment having length 30, the second having length 300, and the third having length 3.

The length of an individual segment should be specified when it is written. RAD classes for development environments such as Delphi and Java usually take care of BLOB segmentation. It will, however, be an issue for embedded and direct-to-API programming and for developers of new or custom connectivity drivers.

For example, the following code fragment inserts a BLOB segment. The segment length is specified in the host variable, *segment_length*:

```
INSERT CURSOR BCINS VALUES (:write_segment_buffer :segment_length);
```

★ The default segment length is 80 bytes, which is optimal for a network packet. The segment length setting does not affect Firebird's performance in processing BLOBS on the server.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Defining segment length

`gpre`, the Firebird precompiler, is used to process SQL statements inside embedded applications. The *segment length* setting, defined for a BLOB column when it is created, is used to determine the size of the internal buffer where the BLOB segment data will be written. This setting tells `gpre` the maximum number of bytes that an application is expected to write to any segment in the column.

Choose the segment length most convenient for the specific application. The largest possible segment length is 32 kilobytes (32,767 bytes).

- ❖ Normally, an application should not attempt to write segments larger than the segment length defined in the table; doing so overflows the internal segment buffer, corrupting memory in the process.

Segment syntax

The following statement creates two BLOB columns, BLOB1, with a default segment size of 80, and BLOB2, with a specified segment length of 512:

```
CREATE TABLE TABLE2
(BLOB1 BLOB,
 BLOB2 BLOB SEGMENT SIZE 512);
```

BLOB subtypes

When you define a BLOB column, you have the option of specifying a *subtype*. A BLOB subtype is a positive or negative integer that identifies the nature of the BLOB data contained in the column. Firebird provides two predefined subtypes for general use: 0 (the default), signifying that a BLOB column contains binary data; and 1, signifying that a BLOB column contains ASCII text.

- *Blob filters* can be written as external routines for creating, handling and translating special subtypes which can be declared to the database using negative integers as unique subtypes.
- Positive integers are reserved for use by Firebird.

The following statement defines three BLOB columns: BLOB1 with subtype 0 (the default), BLOB2 with Firebird subtype 1 (TEXT), and BLOB3 with user-defined subtype -1:

**Symbols** [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

```
CREATE TABLE TABLE2  
  (BLOB1 BLOB,  
   BLOB2 BLOB SUB_TYPE 1,  
   BLOB3 BLOB SUB_TYPE -1);
```

The application is responsible for ensuring that data stored in a BLOB column agrees with its subtype. For example, if subtype -10 denotes a certain data type in a particular application, then the application must ensure that only data of that data type is written to a BLOB column of subtype -10. Firebird does not check the type or format of BLOB data.

Order of clauses

To specify both a default segment length and a subtype when creating a BLOB column, use the SEGMENT SIZE option *after* the SUB_TYPE option, as in the following example:

```
CREATE TABLE TABLE2  
  (BLOB1 BLOB SUB_TYPE 1 SEGMENT SIZE 100 CHARACTER SET DOS437);
```

BLOB filters

BLOB subtypes are used in conjunction with BLOB *filters*. A BLOB filter is a routine that translates BLOB data from one subtype to another. Firebird includes a set of special internal BLOB filters that convert from subtype 0 to subtype 1 (TEXT), and from Firebird system subtypes to subtype 1 (TEXT). In addition to using the internal text filters, programmers can write their own external filters to provide special data translation. For example, an external filter might automatically translate from one bitmapped image format to another.

Associated with every filter is an integer pair that specifies the input subtype and the output subtype. When declaring a cursor to read or write BLOB data, specify FROM and TO subtypes that correspond to the particular BLOB filters. Firebird invokes the filters based on the FROM and TO subtype specified by the BLOB cursor declaration.



For more information about creating external BLOB filters, see [Creating a blob control structure](#) on page 586 in chapter 26.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Viewing BLOBs

The display of the content BLOB subtypes as strings can be specified with SET BLOBDISPLAY in command-line **isql**. Most desktop administration utilities provide the ability to view BLOB data. In some cases, such utilities provide client-side tools to edit and/or convert BLOB data or to view special types through a plug-in viewer.

 For more information about the command-line **isql** tool, see chapter 9, [Interactive SQL Utility \(isql\)](#) (p. 96).

Firebird arrays

Firebird allows you to create homogeneous arrays of most data types. Using an array enables multiple data items to be stored as discrete, multi-dimensional *elements* in a single column. Firebird can perform operations on an entire array, effectively treating it as a single element, or it can operate on an *array slice*, a subset of array elements. An array slice can consist of a single element, or a set of many contiguous elements.

When to use an array type

Using an array is appropriate when:

- The data items naturally form a set of the same data type.
- The entire set of data items in a single database column must be represented and controlled as a unit, as opposed to storing each item in a separate column.
- Each item must also be identified and accessed individually.

Element data types

An array can contain elements of any Firebird data type except BLOB or ARRAY. All of the elements of a particular array are of the same data type.

Defining arrays

Arrays are defined with the CREATE DOMAIN or CREATE TABLE statements. Defining an array column is similar to defining any other column, except that the definition must include the array dimensions. For example, the following statement defines both a regular character column, and a one-dimensional character array column containing four elements:

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

```
EXEC SQL  
CREATE TABLE TABLE1  
(NAME CHAR(10),  
CHAR_ARR CHAR(10)[4]);
```

Array dimensions are always enclosed in square brackets following a column's data type specification.

 For a complete discussion of CREATE TABLE and array syntax, see [Defining columns](#) on p. 315 of chapter 17 and the *Firebird Reference Guide—CREATE TABLE* (ch. 2 p. 104).

To learn more about the flexible data access provided by arrays, refer to the *Embedded SQL Guide* (EmbedSQL.pdf) in the InterBase 6 documentation set, available from Borland.

Multi-dimensional arrays

Firebird supports *multi-dimensional arrays*, arrays with 1 to 16 dimensions. For example, the following statement defines three INTEGER array columns with two, three, and six dimensions respectively:

```
EXEC SQL  
CREATE TABLE TABLE1  
(INT_ARR2 INTEGER[4,5],  
INT_ARR3 INTEGER[4,5,6],  
INT_ARR6 INTEGER[4,5,6,7]);
```

In this example, INT_ARR2 allocates storage for 4 rows, 5 elements in width, for a total of 20 integer elements, INT_ARR3 allocates 120 elements, and INT_ARR6 allocates 840 elements.

❖ Firebird stores multi-dimensional arrays in *row-major order*. Some host languages, such as FORTRAN, expect arrays to be in *column-major order*. In these cases, care must be taken to translate element ordering correctly between Firebird and the host language.

Specifying subscript ranges for array dimensions

Firebird's array dimensions have a specific range of upper and lower boundaries, called *subscripts*. In many cases, the subscript range is implicit. The first element of the array is element 1, the second element 2, and the last is element *n*. For example, the following statement creates a table with a column that is an array of four integers:



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

```
EXEC SQL  
CREATE TABLE TABLE1  
(INT_ARR INTEGER[4]);
```

The subscripts for this array are 1, 2, 3, and 4.

Custom (explicit) subscript boundaries

A custom set of upper and lower boundaries for each array dimension can be explicitly defined when an array column is created. For example, C programmers, familiar with arrays that start with a lower subscript boundary of zero, might want to create array columns with a lower boundary of zero to comply with application code.

When specifying array subscripts for an array dimension, both the lower and upper boundaries of the dimension must be supplied, using the following syntax:

lower:upper

For example, the following statement creates a table with a single-dimension array column of four elements where the lower boundary is 0 and the upper boundary is 3:

```
EXEC SQL  
CREATE TABLE TABLE1  
(INT_ARR INTEGER[0:3]);
```

The subscripts for this array are 0, 1, 2, and 3.

When creating multi-dimensional arrays with explicit array boundaries, separate each dimension's set of subscripts from the next with commas. For example, the following statement creates a table with a two-dimensional array column where each dimension has four elements with boundaries of 0 and 3:

```
EXEC SQL  
CREATE TABLE TABLE1  
(INT_ARR INTEGER[0:3, 0:3]);
```



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Converting data types

Normally, you must use compatible data types to perform arithmetic operations, or to compare data in search conditions. If you need to perform operations on mixed data types, or if your programming language uses a data type that is not supported by Firebird, then data type conversions must be performed before the database operation can proceed.

Implicit type conversions

- In dialect 1, for some expressions, Firebird automatically converts the data to an equivalent data type (an implicit type conversion). The CAST() function can also be used.
- In dialect 3, you can use the CAST() function in search conditions to explicitly translate one data type into another for comparison purposes.
- ♦ Dialect 3 does *not* support implicit string-to-integer conversion.

Firebird supports several types of implicit type conversion. For example, comparing a DATE or TIMESTAMP column to '6/7/2000' causes the string literal '6/7/2000' to be converted implicitly to a DATE entity. An expression mixing integers with scaled numeric types or float types implicitly converts the integer to a like type.

In the following operation:

3 + '1'

- Firebird dialect 1 automatically converts the character "1" to an INTEGER for the addition
- Firebird dialect 3 returns an error

Dialect 3 requires an explicit type conversion:

3 + CAST('1' AS INT)

Both dialects will return an error on the next statement, because Firebird cannot convert the character "a" to an INTEGER:

3 + 'a'



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Explicit type conversions: CAST()

When Firebird cannot do an implicit type conversion, you must perform an explicit type conversion using the CAST() function. Use CAST() to convert one data type to another inside a SELECT statement. Typically, CAST() is used in the WHERE clause to compare different data types. The syntax is:

```
CAST (value | NULL AS data type)
```

Use CAST() to translate the following data types:

- DATE, TIME, or TIMESTAMP data type into a CHARACTER data type.
- CHARACTER data type into a DATE, TIME, or TIMESTAMP data type.
- TIMESTAMP data type into a TIME or DATE data type.
- TIME or DATE data type into a TIMESTAMP data type.

For example, in the following WHERE clause, CAST() is used to translate a CHAR data type, INTERVIEW_DATE, to a DATE data type in order to compare against a DATE data type, HIRE_DATE:

```
... WHERE HIRE_DATE = (CAST(INTERVIEW_DATE AS DATE));
```

In the next example, CAST() is used to translate a DATE data type into a CHAR data type:

```
... WHERE CAST(HIRE_DATE AS CHAR) = INTERVIEW_DATE;
```

You can use CAST() to compare columns with different data types in the same table, or across tables.



For more information, see [Casting between time and date types](#) on page 254.

Changing column and domain definitions

In both dialects, you can change the data type of a column in tables and domains.

Altering columns in tables

Use the ALTER COLUMN clause of the ALTER TABLE statement. For example:

```
ALTER TABLE table1 ALTER COLUMN field1 TYPE char(20);
```



The ALTER COLUMN clause of the ALTER TABLE expression is useful for migration.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Altering domains

The ALTER DOMAIN statement allows the data type of a domain to be changed. For example:

```
ALTER DOMAIN domain1 TO domain2;
```

Constraints on altering data types

- 1 Firebird does not allow the data type of a column or domain to be altered in a way that might result in data loss. For example, they do not allow you to change the number of characters in a column to be smaller than the largest value in the column.
- 2 Converting a numeric data type to a character type requires a minimum length for the character type as listed below.

TABLE 13-11 Minimum character lengths for numeric conversions

data type	Minimum length for converted character type
Decimal	20
Double	22
Float	13
Integer	11
Numeric	22
Smallint	6



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

CHAPTER 14

Databases

The chapter provides an introduction to creating and working with databases.



RELATED TOPICS

- Details about defining [Firebird domains](#), [Firebird Generators](#), [Character Sets and Collation Orders](#), [Tables](#), [Indexes](#) and [Views](#) are discussed in the five chapters following this
- For further topics regarding maintenance and management of database files on your servers, see chapter 20, [More About Databases](#) (p. 375)

What you should know

Before creating the database, you should know:

- Where to create the database.
- ★ Users who create databases need to know only the logical names of the available devices in order to allocate database storage. Only the system administrator needs to be concerned about physical storage (disks, disk partitions, operating system files).
- The structures of the most frequently accessed tables to be contained.
- The *row size* of each table, which affects the database page size you choose. A record that is too large to fit on a single page requires more than one page fetch to read or write to it, so access can be optimized by choosing a page size that can comfortably accommodate one row or simple row multiples of your most frequently accessed tables.
- The *number of rows* that your main tables can be predicted to accommodate over time. This factor also affects the page size because the number of pages affects the depth of the index tree. Larger

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

page size reduces the overall tally of data and index pages. Firebird operates more efficiently with a shallow index tree.

- How large you expect the database to grow and an approximation of the growth rate. This information will affect whether you create a multi-file database and the number of files you start with.
- The number of users that will be accessing the database.

Database object naming conventions

When an applications programmer or end user creates a database object or refers to it by name, case is unimportant. The following limitations on naming database objects must be observed:

- Start each name with an alphabetic character (A-Z or a-z).
- Restrict object names to 31 characters.
 - ❖ Some objects, such as constraint names, are restricted to 27 bytes in length
- Allowable characters for database file names—as with all metadata objects in Firebird—include dollar signs (\$), underscores (_), 0 to 9, A to Z, and a to z..
- Observe uniqueness requirements within a database:
 - In all cases, objects of the same type—all tables, for example—*must* be unique.
 - Except for column identifiers, object names must also be unique within the database. (Column identifiers must be unique within a table).
- Avoid the use of reserved words, diacritic characters and case-sensitivity in database object identifiers. In dialect 1, they cannot be used at all. However, if there is an essential reason to do so, in dialect 3 you can delimit “illegal” identifiers using pairs of double-quote symbols. Details follow.

Delimited identifiers

In dialect 3 database, Firebird supports the ANSI SQL convention for optionally delimiting identifiers. To use reserved words, diacritic characters, case-sensitive strings, or spaces (except for trailing spaces) in an object name, enclose the name in double quotes. It is then a *delimited identifier*. Delimited identifiers must always be referenced in double quotes.

Names enclosed in double quotes are case sensitive. For example:

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

SELECT "CodAR" FROM MyTable

is different from:

SELECT "CODAR" FROM MyTable

 For more information about naming database objects with CREATE or DECLARE statements, refer to the chapters listed in the introduction to this chapter.

Database file-naming conventions

The established convention for naming InterBase® database files on any platform is to apply the three-character suffix ".gdb" to the primary file and to name secondary files ".g01", ".g02", etc. This is only a convention—a Firebird database file can have any extension, or no extension at all.

Because of known problems on XP servers, involving the SystemRestore feature actively targeting files with the suffix ".gdb", many developers choose to use the suffix ".fdb". In time, this may become the convention for Firebird; however, it remains quite arbitrary.

 The name of the security database, **isc4.gdb**, must not be changed. Unfortunately, Firebird 1 has no workaround for this limitation.

Creating a database

You can create a database in **isql** or your favorite GUI database administration tool with an interactive CREATE DATABASE command or with a CREATE DATABASE statement in an **isql** script file.

 Although you *can* create, alter, and drop a database interactively, it is preferable to use a data definition file (DDL script) because it provides a record of the structure of the database. It is easier to modify a source file than it is to start over by retyping interactive SQL statements.

 For help with creating and running scripts, see chapter 12, [Writing and Running Scripts](#) (p. 222).



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Using CREATE DATABASE

CREATE DATABASE establishes a new database and populates its system tables, which are the metadata that describe the internal structure of the database.

CREATE DATABASE must be the first statement in a DDL script. It must be successfully committed before it is possible to create contained database objects such as domains, tables, views, and indexes.

- ❖ In DSQL, CREATE DATABASE can be executed only with EXECUTE IMMEDIATE. The database handle and transaction name, if present, must be initialized to zero prior to use.

The CREATE DATABASE statement requires a file specification to name and locate the new database file. If you are logged in as SYSDBA then SYSDBA will own the new database. The CREATE DATABASE statement optionally allows you to supply parameters to specify

- a user name and a password for the database owner
- the page size for the new database
- a default character set
- secondary files to expand the database beyond the default single file

Syntax

The syntax for CREATE DATABASE is:

```
CREATE {DATABASE | SCHEMA} 'filespec'  
[USER 'username' [PASSWORD 'password']]  
[PAGE_SIZE [=] int]  
[LENGTH [=] int [PAGE[S]]]  
[DEFAULT CHARACTER SET charset]  
[<secondary_file>];  
  
<fileinfo> = LENGTH [=] int [PAGE[S]]  
| STARTING [AT [PAGE]] int [<fileinfo>]  
  
<secondary_file> = FILE 'filespec' [<fileinfo>] [<secondary_file>]
```

- ★ Use single quotes to delimit strings such as file names, user names, and passwords.



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

Creating a single-file database

Although there are many optional parameters, CREATE DATABASE requires only one parameter, *filespec*, which is the new database file specification. The file specification contains the device name, path name, and database name.

TABLE 14–1 File specification for a local database

Windows: 'drive:\path\mydatabase.gdb'

Linux/UNIX: 'localhost:/path/mydatabase.gdb'

TABLE 14–2 File specification for a remote database—

includes the name of the host machine (*server_name*) and depends on the protocol:

TCP/IP

Windows: 'server_name:drive:\path\mydatabase.gdb'

Linux/UNIX: 'server_name:/path/mydatabase.gdb'

NetBEUI

(only if host machine is '\\server_name\drive:\path\mydatabase.gdb'
Windows NT/2K/XP)

★ You can use either forward slashes (/) or backslashes (\) as directory separators. Firebird automatically converts either type of slash to the appropriate type for the server operating system. If *filespec* contains spaces, the entire string must be enclosed in single quotes.

By default, a database is created as a single file, called the *primary file*. The following example creates a single-file database, named **employee.gdb**, in the current directory.

```
CREATE DATABASE 'employee.gdb' ;
```

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z****Specifying file size for a single-file database**

You can optionally specify a file length, in pages, for the primary file. For example, the following statement creates a database that is stored in one 10,000-page-long file:

```
CREATE DATABASE 'employee.gdb' LENGTH 10000;
```

If the database grows larger than the specified file length, Firebird extends the primary file beyond the LENGTH limit until the filesystem size limit for writable file is reached or disk space runs out. To avoid this, you can store a database in more than one file, called a *secondary file*.

- ❖ It is NOT recommended to specify a LENGTH for the primary file if you are creating a single-file database. Use it only if defining one or more secondary files in the same statement.

Multi-file databases

Operating system file systems limit the size of writable files. The limit varies from file system to file system but, in general, the limit will be either 2 Gb or 4 Gb. The InterBase® family of databases, from which Firebird was extended, provides the ability to set up databases as an extensible suite of files. Single-file databases can also be ported to multiple files very simply, using the **gbak** utility.

- ❖ For instructions on porting a single file database to multiple files using **gbak**, see [Multiple-file restores](#) on p. 398 of chapter 21.

A traditional problem with InterBase® databases has been that, when a database approached the file system size limit of the primary database file, with no secondary files defined, corruption began to occur, with new data overwriting current data. A similar problem occurred if a database exhausted its secondary file capacity. The problem could go undetected for some time, until the database became unusable. At that point, corruption would be so severe that large blocks of data would be unrecoverable.

Firebird has corrected this problem, to the extent that it will not permit writes to a database that has reached the limit of its capacity. Corruption of existing data is thus prevented, although any outstanding writes will be lost.

It is the responsibility of the database administrator to monitor database growth and ensure that a database always has ample capacity for extension.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Creating a multi-file database

A multi-file database consists of a *primary file* and one or more *secondary files*. Any secondary file can be assigned to a different disk from that of the main database, as long as the disk is physically under the control of the host machine that is running Firebird server.

- ❖ Secondary files can not be assigned to network disks or filesystems.
- ❖ File specifications for secondary files cannot include a host name. Whenever possible, create databases locally, in order to be able to accommodate adequate valid file specifications for each file.

It is not possible to specify which data shall be stored in any particular secondary file because Firebird manages the physical on-disk structures automatically. With a multi-file database, Firebird writes to the primary file until it has filled the specified number of pages, then proceeds to fill the next specified secondary file.

Size parameters for multiple files

When you define a secondary file, you can choose to specify its size in database pages (LENGTH), or you can specify the initial page number of the following file (STARTING AT). Firebird always treats the final file of a multifile database as dynamically sizeable: it increases the size of the last file as needed.

- ★ Although specifying a LENGTH for the final file does not return an error, a LENGTH specification for the last—or only—file of a database is meaningless.

Specifying a secondary file using LENGTH

The LENGTH parameter specifies the number of database pages for the file. The eventual maximum file size is then the number of pages times the page size for the database.

- ★ Because file-naming conventions are platform-specific, for the sake of simplicity, none of the examples below includes the device and path name portions of the file specification.

The following example creates a database with a primary file and three secondary files. The primary file and the first two secondary files are each 10,000 pages long:

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

```
CREATE DATABASE 'employee.gdb' LENGTH 10000
FILE 'employee2.gdb' LENGTH 10000 PAGES
FILE 'employee3.gdb' LENGTH 10000 PAGES
FILE 'employee4.gdb';
```

Specifying the starting page number of a secondary file

If you do not declare a length for a secondary file, then you must specify a starting page number. STARTING AT specifies the beginning page number for a secondary file. The PAGE reserved word is optional.

```
CREATE DATABASE 'employee.gdb'
FILE 'employee2.gdb' STARTING AT PAGE 10001
FILE 'employee3.gdb' STARTING AT 20001
FILE 'employee4.gdb' STARTING AT 30001;
```

★ If you specify a STARTING AT parameter that is inconsistent with a LENGTH parameter for the previous file, the LENGTH specification takes precedence.

You can specify a combination of length and starting page numbers for secondary files. The following example produces exactly the same results as the previous one, but uses a mixture of LENGTH and STARTING AT:

```
CREATE DATABASE 'employee.gdb'
FILE 'employee2.gdb' STARTING AT 10001 LENGTH 10000 PAGES
FILE 'employee3.gdb' LENGTH 10000 PAGES
FILE 'employee4.gdb';
```

Specifying user name and password

If provided, the user name and password are checked against valid user name and password combinations in the security database on the server where the database will reside. Only the first 8 characters of the password are significant.

❖ Where Windows client applications must create their databases through a remote client connection, the user name and password are *not* optional.

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

The following statement creates a database with a user name and password:

```
CREATE DATABASE 'employee.gdb' USER 'SALES' PASSWORD 'mycode';
```

Specifying database page size

The default database page size is 4096 bytes. It can be overridden using the PAGE_SIZE parameter. PAGE_SIZE can be 1024, 2048, 4096, 8192 or 16384 bytes. The next statement creates a single-file database with a page size of 8,192 bytes:

```
CREATE DATABASE 'employee.gdb' PAGE_SIZE 8192;
```

WHEN TO INCREASE PAGE SIZE

Increasing page size can improve performance for several reasons:

- Indexes work faster because the depth of the index is kept to a minimum.
- Keeping large rows on a single page is more efficient. (A row that is too large to fit on a single page requires more than one page fetch to read or write to it.)
- BLOB data is stored and retrieved more efficiently when it fits on a single page. If an application frequently stores large BLOB columns (between 1K and 2K), use a page size of at least 4,096 bytes.

If most transactions involve only a few rows of data, a smaller page size might be appropriate, since less data needs to be passed back and forth and less memory is used by the disk cache.

CHANGING PAGE SIZE FOR AN EXISTING DATABASE

To change a page size of an *existing* database, follow these steps:

- 1 Back up the database.
- 2 Restore the database using the PAGE_SIZE option to specify a new page size.



For more detailed information, see [Arguments for qbak -C\[reate\] and -R\[estore\]](#) on p. 399 of chapter 21.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Specifying the default character set

DEFAULT CHARACTER SET allows you to optionally set the default character set for the database. The character set determines:

- the characters that can be used in CHAR, VARCHAR, and BLOB text columns.
- the default collation order that is used in sorting a column.

Choosing a default character set is useful for all databases, even those where international use is not an issue. Choice of character set determines whether transliteration among character sets is possible. For example, the following statement creates a database that uses the ISO8859_1 character set, commonly used to support European languages:

```
CREATE DATABASE 'employee.gdb'  
  DEFAULT CHARACTER SET 'ISO8859_1';
```



For a list of the international character sets and collation orders that Firebird supports, see *Firebird Reference Guide—Character Sets and Collation Orders* (ch. 4 p. 249) and chapter 16, *Character Sets and Collation Orders* in this volume.

Using CHARACTER SET NONE

If you do not specify a default character set, the character set defaults to NONE. Using CHARACTER SET NONE means that there is no character set assumption for columns; data are stored and retrieved just as you originally entered them. You can load any character set into a column defined with NONE, but you cannot load that same data into another column that has been defined with a specific character set. No transliteration will be performed between the source and destination character sets, so in most cases, errors will occur during the attempted assignment.

For example:

```
CREATE TABLE MYDATA (PART_NUMBER CHARACTER(30) CHARACTER SET NONE);  
SET NAMES LATIN1;  
INSERT INTO MYDATA (PART_NUMBER) VALUES ('à');  
SET NAMES DOS437;  
SELECT * FROM MYDATA;
```

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

The data ("à") is returned just as it was entered, without the à being transliterated from the input character (LATIN1) to the output character (DOS437). If the column had been set to anything other than NONE, the transliteration would have been attempted and would have failed..

Connecting to a database

If you have just finished creating a database, you will be automatically connected to it until you terminate your connection. At other times, you will need to connect to a database in order to work with it. The means you use to connect depends on the client environment through which you make the connection.



FOR INFORMATION ABOUT CONNECTING THROUGH THE COMMAND-LINE UTILITIES:

- **isql**—see [Connecting to a database](#) on p. 154 of chapter 10
- **gfix**—see [Using gfix](#) on p. 469 of chapter 24
- **gstat**—see [gstat command-line tool](#) on p. 452 of chapter 23
- **gsec**—see [Maintaining the security database](#) on p. 421 of chapter 22
- see chapter 12, [Writing and Running Scripts](#) (p. 222) for the CONNECT string to use in scripts

The Windows and Linux GUI desktop programs provide transparent interfaces for the connection and login procedures.

Most RAD application development tools provide API classes encapsulating the CONNECT statement and its parameters.

Programmers working directly with the API or writing embedded applications should refer to the following for detailed information about the CONNECT statement and the database parameter block (DPB):

- Firebird Reference Guide (companion volume to this book)
- latest Firebird release notes
- API Guide ([APIGuide.pdf](#)) and Embedded SQL Guide ([EmbedSQL.pdf](#)) in the InterBase® 6 documentation set, obtainable through Borland merchandising outlets



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Getting exclusive access to a database

Several database-level maintenance and administrative tasks require exclusive access to the database. To get exclusive access, you need to

- 1 connect to the database as SYSDBA, as the database owner or (on Linux/UNIX) as a user with *root* privileges
- 2 perform an explicit shutdown of the database. Once the database is in the shut down state, the server will go into a state where “ordinary” users can not log in.

❖ “Shutting down the server” and “shutting down a database” are not the same thing. When you shut down the server, it is usually desirable to shut down individual databases first.

BOOK For instructions on shutting down a database, and the options available, see [Shutting down a database](#) on p. 471 of chapter 24.

About exclusive access

Even when a database is shutdown, the user who performed it—the SYSDBA user, the owner or the Linux/UNIX root user—can log on multiple times. Because truly exclusive access to the database cannot be guaranteed, it gets down to being a question of discipline in your organisation to ensure that all of the people who have access to these privileges fully understand the need to avoid stepping on one another’s toes when a crucial administrative task is to be performed.

‘Single-user mode’

Other than the exclusive access that a privileged user may obtain by means of a database shutdown, there is no “single-user” mode in Firebird. Even the local connection capability for a Windows machine is not truly “single-user”, since the local user can make multiple connections to any database.

In order for the SYSDBA or the owner to acquire the level of exclusive access required for certain tasks, that user must be the only user logged in AND should not have any competing transactions active. This error message will appear if you attempt to perform a task requiring exclusive access whilst another user is still connected or another transaction is in progress:

```
OBJECT database_name IS IN USE
```

**Symbols** [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

This message sometimes baffles people, since it can appear when you are certain that everyone else has logged out. Conditions can occur when the server knows best in this regard—for example:

- it may still detect a connection that was earlier broken abnormally by a client or a network fault
 - ❖ Sometimes, client software does not do a good job of cleaning up connections when users exit from programs.
- during the pre-shutdown waiting period, if another privileged user instance (owner, SYSDBA or *root* user) was connected, Firebird ignores them in calculating the connections or transactions that are watched for log-out
- you may have other transactions active, e.g. an isql session that you have not terminated with EXIT or QUIT

As a last resort, you may need to shut down the server, restart it, log in to the database as a privileged user and then perform another shutdown.

Restoring multiple-user access

The `-o[nline]` switch of `gfix` rescinds a shutdown that is currently in effect:

```
gfix -sh -o db_name
```

More about databases

 For further topics regarding maintenance and management of database files on your servers, see chapter 20, [More About Databases](#) (p. 375). See also

- chapter 21, [Database Backup and Restore](#)
- chapter 22, [Managing Security](#)
- chapter 23, [Server and Database Statistics](#)
- chapter 24, [Housekeeping & Repair: gfix](#), re repairing corrupted data structures and configuring special features,



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
CHAPTER 15

Domains and Generators

Firebird domains

Domains in Firebird are akin to the concept of “user-defined data types”. Although it is not possible to create a new data type, with a domain you can package a set of attributes with those of an existing data type, give it an identifier and, thereafter, use it in place of the data type parameter to define columns for any table.

Domain definitions are global to the database—all columns in any table which are defined with a particular domain that is unmodified by local overrides will have completely identical attributes.

The benefits for objectivizing data definition are obvious. For example, suppose your design calls for a number of small tables where you want to store the text descriptions of enumerated sets—“type” tables—account types, product types, subscriptions types, etc. You have decided that each member of each of these sets will be keyed on a three-character upper-case identifier which points to a character field having a maximum of 25 characters.

All that is required is to create two domains:

- the domain for the pointer will be a CHAR(3) with two attributes added—a NOT NULL constraint because you are going to use it for primary and lookup keys and a CHECK constraint to enforce upper-case. For example,

```
CREATE DOMAIN Type_Key AS CHAR( 3 ) NOT NULL  
CHECK ( VALUE = UPPER( VALUE ) );
```

- the description domain will be a VARCHAR(25). You want it to be non-nullable because the tables you want to use it in are control tables:

```
CREATE DOMAIN Type_Description AS VARCHAR( 25 ) NOT NULL
```

Once you have these domains defined, all of your type-lookup tables can have similar definitions; and all tables which store lookup keys to these tables will use the matching domain for the key column.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Domain overrides

Columns based on a domain definition inherit all attributes of the domain, which can be:

- data type (required)
 - a default value for inserts
 - NULL status
 - CHECK constraints
 - character set (for character and BLOB columns only)
 - collation order (for character columns only)
- ❖ You cannot apply referential integrity constraints to a domain.

When defining table columns using a domain, you can override attributes inherited from the domain, by replacing an inherited attribute with an equivalent attribute using a parameter specific to the column you are defining; and by adding further attributes.

- ❖ It is not possible to override the *data type* attribute.

Syntax for CREATE DOMAIN

```
CREATE DOMAIN domain [AS] <datatype>
[DEFAULT {literal | NULL | USER}]
[NOT NULL] [CHECK (<dom_search_condition>)]
[CHARSET {charset | NONE}]
[COLLATE collation];
```

Using CREATE DOMAIN

Domain identifier

When you create a domain in the database, you must specify an identifier for the domain which is globally unique in the database.

- ★ Developers often use a special prefix or subscript in domain identifiers, to facilitate self-documentation. For example,

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

CREATE DOMAIN D_TYPE_IDENTIFIER...

CREATE DOMAIN D_TYPE_DESCRIPTION...

Data type for the domain

The *data type* is the only required attribute that must be set for the domain—all other attributes are optional. It specifies the SQL data type that will apply to column defined using the domain.

- ❖ You cannot override the data type when using the domain in a column definition.

The general categories of SQL data types include:

- Character types
- Integer types
- Exact numeric, including NUMERIC(18,0)—the 64-bit integer type
- Floating-point numeric
- Date and time types
- A BLOB datatype to represent unstructured binary data, such as graphics and digitized voice.
- Multi-dimensional arrays of any type except BLOB and array

The syntax for specifying the datatype is:

```
<datatype> = SMALLINT  
| INTEGER  
| FLOAT  
| DOUBLE PRECISION  
| {DECIMAL | NUMERIC} [(precision [, scale])]  
| {DATE | TIME | TIMESTAMP}  
| {CHAR | CHARACTER | CHARACTER VARYING | VARCHAR}  
[(int)] [CHARACTER SET charname]  
| BLOB [SUB_TYPE int SEGMENT SIZE int]  
| {NCHAR | NATIONAL CHARACTER | NATIONAL CHAR} [VARYING] [(int)]  
  
<array_dim> = [x:y [, x1:y1 ...]]
```



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- ❖ The outermost (boldface) brackets must be included when declaring arrays.

For more information about data types, see chapter 13, [Firebird Data Types](#) (p. 231) and the list in *Firebird Reference Guide*— [Supported Datatypes](#) (ch. 5 p. 254).

The following statement creates a domain that defines an array of CHARACTER datatype:

```
CREATE DOMAIN DEPTARRAY AS CHAR(31) [4:5];
```

The next statement creates a BLOB domain with a text subtype that has an assigned character set:

```
CREATE DOMAIN DESCRIPT AS BLOB SUB_TYPE TEXT SEGMENT SIZE 80  
CHARACTER SET SJIS;
```

The DEFAULT attribute

Use this attribute if you want to have Firebird write a default value in INSERT statements whenever the statement does not specify a value explicitly. Defaults can save time and error during data entry. For example, a possible default for a DATE column could be today's date, or to write the CURRENT_USER context variable into a UserName column.

Default values can be:

- *a constant*: The default value is a user-specified string, numeric value, or date value.
- *NULL*: If the user does not enter a value, a NULL value is entered into the column.
- a Firebird *predefined date literal*
- *USER*, *CURRENT_USER* or *CURRENT_ROLE*: The default is the name of the current user or role.
- ❖ If your operating system supports the use of multi-byte characters in user names, or you have used a multi-byte character set when defining roles, then any column into which these defaults will be stored must be defined using a compatible character set.
- Other context variables (CURRENT_DATE, CURRENT_TIMESTAMP, CURRENT_TIME)

When defaults won't work

It is a common mistake to assume that a default value will be used whenever Firebird receives NULL in a defaulted column. When relying on defaults, it must be understood that a default will be applied

- only upon insertion of a new row



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

AND

- only if the INSERT statement does not include the defaulted column in its column list
- ❖ If your application includes the defaulted column in the INSERT statement and sends NULL in the values list, then NULL will be stored, regardless of any default defined.

Overriding DEFAULT

A default set in a domain definition can be overridden in a CREATE TABLE or ALTER TABLE column definition using the domain.

Example

In the following example, the first statement creates a domain with USER named as the default. The next statement creates a table that includes a column, ENTERED_BY, based on the USERNAME domain.

```
CREATE DOMAIN USERNAME AS VARCHAR(20) DEFAULT CURRENT_USER;
CREATE TABLE ORDERS (ORDER_DATE DATE, ENTERED_BY USERNAME,
ORDER_AMT DECIMAL(8,2));
INSERT INTO ORDERS (ORDER_DATE, ORDER_AMT)
VALUES ('1-MAY-93', 512.36);
```

The INSERT statement does not include a value for the ENTERED_BY column, so Firebird automatically inserts the user name of the current user, JSmith:

```
SELECT * FROM ORDERS;
1-MAY-93 JSWILL 512.36
```

The NOT NULL attribute

Include this attribute in the domain if you want to force all columns created with this domain to contain a value. NULL—which is not a value, but a *state*—will always be disallowed on any column bearing this attribute.

- ❖ Important considerations with respect to including the NOT NULL attribute:
 - You cannot override the NOT NULL attribute on a domain. Consider the benefit of *not* including it in the domain's attributes, thereby leaving it as an option to add the attribute when columns are defined.

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

- If NULL is being included in the domain as a default value, be sure not to create contradictory constraints by also assigning the NOT NULL attribute.

CHECK data conditions

The CHECK constraint provides wide scope for providing domain attributes that restrict the content of data which can be stored in columns using the domain. The CHECK constraint sets a search condition (*dom_search_condition*) that must be true before data can be accepted into these columns.

Syntax for CHECK constraints

```
<dom_search_condition> =
  VALUE <operator> <val>
  | VALUE [NOT] BETWEEN <val> AND <val>
  | VALUE [NOT] LIKE <val> [ESCAPE <val>]
  | VALUE [NOT] IN (<val> [, <val> ...])
  | VALUE IS [NOT] NULL
  | VALUE [NOT] CONTAINING <val>
  | VALUE [NOT] STARTING [WITH] <val>
  (<dom_search_condition>)
  NOT <dom_search_condition>
  <dom_search_condition> OR <dom_search_condition>
  <dom_search_condition> AND <dom_search_condition>

<operator> = {= | < | > | <= | >= | !< | !> | <> | !=}
```

The VALUE keyword

VALUE is a placeholder for any constant or variable value or expression result which would be submitted through SQL for storing in a column defined using the domain. The purpose of the CHECK condition is to compare VALUE with the attributes defined in the search criteria and either validate or reject VALUE.

- ❖ If NULL values are allowed, they must be included in the CHECK constraint, as in the following example:

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

```
CHECK ((VALUE IS NULL) OR (VALUE > 1000));
```

The next statement creates a domain where VALUE must be greater than 1,000:

```
CREATE DOMAIN CUSTNO  
AS INTEGER  
CHECK (VALUE > 1000);
```

The following statement creates a domain that must have a positive value greater than 1,000. If the INSERT statement does not present a VALUE, the column will be assigned the default value of 9,999.

```
CREATE DOMAIN CUSTNO  
AS INTEGER  
DEFAULT 9999  
CHECK (VALUE > 1000);
```

The next statement restricts VALUE to being one of four specific values:

```
CREATE DOMAIN PRODTYPE  
AS VARCHAR(12)  
CHECK (VALUE IN ('software', 'hardware', 'other', 'N/A'));
```

When a problem cannot be solved using comparisons, you can instruct the system to search for a specific pattern in a character column. For example, the next search condition allows only cities in California to be entered into columns that are based on the CALIFORNIA domain:

```
CREATE DOMAIN CALIFORNIA  
AS VARCHAR(25)  
CHECK (VALUE LIKE '%, CA');
```

❖ Important considerations with respect to CHECK constraints:

- A CHECK constraint cannot refer to any other domain or column name.
- A domain can have only one CHECK constraint—multiple conditions can be ANDed or ORed within this single predicate.
- A domain's CHECK constraint cannot be overridden by one declared during column definition. However, a column can *add* an additional CHECK constraint to that inherited from the domain.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

The CHARSET/CHARACTER SET attribute

For applications that need to be concerned about character sets, declaring domains for all of your text columns (CHAR, VARCHAR, BLOB SUB_TYPE 1 and arrays of character types) can be a very elegant way to deal with it.

If you need your text columns to be CHARSET ISO8859_1, you can simplify the declaration of your text domain attributes even more—refer to [NCHAR\(n\)](#) or [NATIONAL CHAR\(n\)](#) and [NCHAR VARYING\(n\)](#) in chapter 13, [Firebird Data Types](#).

Column-level override

If a different character set is set in the CREATE TABLE or ALTER TABLE statement for the column where the domain is used, the column-level setting will override the one inherited from the domain.

- ❖ For details of character sets available, refer to the *Firebird Reference Guide*—[Character Sets and Collation Orders](#) (ch. 4 p. 249).

The COLLATE attribute

With a COLLATE clause in a domain creation statement, you can specify a particular collation order for columns defined as text of CHAR or VARCHAR type. You must choose a collation order that is supported for the column's declared, inherited or implied character set.

- ❖ The character set is either the default character set for the entire database, or one which you have specified in the CHARSET [CHARACTER SET] sub-clause of the data type clause in your domain definition.

Column-level override

If a different collation order is set in the CREATE TABLE or ALTER TABLE statement for the column where the domain is used, the column-level setting will override the one inherited from the domain.

In the following statement, the domain, TITLE, overrides the database default character set, specifying a DOS437 character set with a PDOX_INTL collation order:

```
CREATE DOMAIN TITLE AS  
CHAR(50) CHARACTER SET DOS437 COLLATE PDOX_INTL;
```



Symbols [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

For a list of the collation orders available for each character set, see chapter 16, [Character Sets and Collation Orders](#) (p. 301).

Altering domains

The DDL statement ALTER DOMAIN can be used to change any aspect of an existing domain except its NOT NULL setting. Changes that you make to a domain definition affect all column definitions based on the domain that have not been overridden at the table level.

- ❖ The only way to “change” the NOT NULL setting of a domain is to drop the domain and recreate it with the desired combination of features.

The ALTER DOMAIN statement

Privileges A domain can be altered by its creator, the SYSDBA user or (on Linux/UNIX) any user with operating system root privileges.

ALTER DOMAIN allows you to:

- Rename the domain
- Modify the data type
- Drop an existing default value.
- Set a new default value.
- Drop an existing CHECK constraint.
- Add a new CHECK constraint.

Syntax

```
ALTER DOMAIN { name | old_name TO new_name } {  
  [SET DEFAULT {literal | NULL | USER}]  
  | [DROP DEFAULT]
```

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

```
| [ADD [CONSTRAINT] CHECK (<dom_search_condition>)]  
| [DROP CONSTRAINT]  
| new_col_name  
| TYPE data_type  
};
```

The following statement sets a new default value for the CUSTNO domain:

```
ALTER DOMAIN CUSTNO SET DEFAULT 9999;
```

The following statement changes the name of the CUSTNO domain to CUSTNUM:

```
ALTER DOMAIN CUSTNO TO CUSTNUM;
```

The following statement changes the data type of the CUSTNUM domain to CHAR(20):

```
ALTER DOMAIN CUSTNUM TYPE CHAR(20);
```

- ❖ With the TYPE clause of ALTER DOMAIN, it is not possible to make any type conversion that could result in data loss. For example, the number of characters in a column could not be made smaller than the size of the largest value in the column.

Converting a numeric data type to a character type requires a minimum length for the character type as listed below:

TABLE 15–1 Minimum character lengths for numeric conversions

data type	Minimum length for converted character type
Decimal	20
Double	22
Float	13
Integer	11



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

TABLE 15–1 Minimum character lengths for numeric conversions

data type	Minimum length for converted character type
Numeric	22
Smallint	6



For a list of valid data type conversions, refer to the relevant data types in chapter 13, [Firebird Data Types](#).

Dropping a domain

DROP DOMAIN removes an existing domain definition from a database, provided the domain is not currently used for any column definition in the database.

- ★ To prevent failure, delete the columns based on the domain with ALTER TABLE before executing DROP DOMAIN.

Privileges A domain can be dropped by its creator, the SYSDBA user or (on Linux/UNIX) any user with operating system root privileges.

Syntax

```
DROP DOMAIN name;
```

The following statement deletes a domain:

```
DROP DOMAIN COUNTRYNAME;
```



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Firebird Generators

A *generator* is a Firebird mechanism that creates and maintains a series of unique sequential numbers. The generation of a new number from a generator is the only operation in Firebird that works outside the control of any user transaction. Provided the generation of a series is not subjected to human interference, each generated value is guaranteed to be unique.

Generators vs autoincrement types

Unlike autoincrement or “identity” types in other database management systems, that automatically insert unique serial numbers into primary key or other columns, a generator is not a data type and does not provide any internal capability to populate a column.

How generators are used

Use of a generator to populate a serial column independently of client operations requires these two steps:

- 1 creation of a generator by simply giving it a unique identifier, e.g.

```
CREATE GENERATOR MyGenerator;
```

- 2 applying a BEFORE INSERT trigger to the table, to make the table fetch a new number from the generator into the column which uses it, e.g.

```
CREATE TRIGGER Gen_MyGenerator FOR Customer
ACTIVE BEFORE INSERT POSITION 0 AS
BEGIN
  IF (NEW.PKEY IS NULL) THEN
    NEW.PKEY = GEN_ID(MyGenerator, 1);
END
```

At no point is the generator explicitly linked to the column or columns it will populate. Use of the generator is a matter of programming. In fact, there is no absolute requirement to use a trigger at all. Many client application interfaces are designed to acquire the generated value from the database through a function and assign it through client data structures during interactive data entry operations.

A single generator may even be shared as the source of unique primary keys for several tables.

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

Generators are commonly used to produce unique values for insertion into a column that is used as a PRIMARY KEY or a serial number.

For example, a programmer writing an application to log and track invoices may want to ensure that each invoice number entered into the database is unique. The programmer can use a generator to create the invoice numbers automatically, rather than writing specific application code to accomplish this task.

A generator is global to the database where it is declared. Any number of generators can be defined for a database, provided each generator has a unique name. Any transaction that activates the generator can fetch a new number and be guaranteed that it is unique and not capable of being re-used—Firebird does not (and can not) assign duplicate generator values across transactions.

Creating generators

The CREATE GENERATOR statement declares a generator to the database and sets its starting value to zero.

- ★ If you want to set the starting value for the generator to a number other than zero, use a separate SET GENERATOR statement to specify the new value.

Syntax

```
CREATE GENERATOR name;
```

The following statement creates the generator, EMPNO_GEN:

```
CREATE GENERATOR EMPNO_GEN;
```

Generators and data types

Once a Firebird generator reaches the limit of the 32-bit integer range, it begins returning a 64-bit value. The generator will “wrap around” and start again from zero after $2^{31} - 1$ invocations, assuming an increment of 1.

Columns and procedure variables defined to store the output of a generator can be defined as SMALLINT, INTEGER or NUMERIC(18,0), according to requirements, and Firebird will perform downward-casting required.

- ❖ It is the designer's responsibility to ensure that the choice of integer type will not give rise to overflow errors during the life of the application.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Applications can use a 32-bit or 64-bit integer variable to hold the value returned by a generator. The judgment of which type to use to avoid overflow will depend upon the maximum range to which your series is expected to extend.

- ❖ A client using dialect 1 receives only the least significant 32 bits of the updated generator value, but the entire 64-bit value is incremented by the engine even when returning a 32-bit value to a client.

Using a generator

For retrieving the next value in a generator's series, Firebird provides the GEN_ID() function:

```
GEN_ID(generator_name, increment)
```

GEN_ID() takes two arguments: *generator_name* is the identifier of a generator which exists in the database and *increment* is a signed integer specifying the increment between the previous generated number and this one—sometimes referred to as the *step interval*.

GEN_ID() can be called from within a trigger or a stored procedure, or it can be retrieved by an application through a DSQL query. For example, the following statement uses GEN_ID() to call the generator G to increment a purchase order number in the SALES table by one:

```
INSERT INTO SALES (PO_NUMBER) VALUES (GEN_ID(G,1));
```

Fetching the previous value

A step interval of 0 will fetch the previous value without incrementing the generator, e.g.

```
...
DECLARE VARIABLE old_gen_value NUMERIC(18,0);
...
old_gen_value = GEN_ID(MyGenerator, 0);
...
```

- ❖ Do not write application code that depends on GEN_ID(*generator_name*, 0) to make assumptions about the state of any data that are outside the control of your current transaction. This is a very unsafe strategy in a multi-user environment.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Using GEN_ID() to decrement the generator

It is also possible (but not recommended) to decrement the generator's value by passing a negative step interval, e.g.

```
...
DECLARE VARIABLE older_gen_value NUMERIC(18,0);
...
older_gen_value = GEN_ID(MyGenerator, -1);
...
```

- ❖ Never use `GEN_ID()` with a negative step interval in either application or server code. It is one of the few completely predictable ways to corrupt data in Firebird.

Retrieving a generator value into an application

The following DSQL statement will return the new value of a generator as a singleton row of one INT64 field:

```
SELECT GEN_ID(generator_name, increment) FROM RDB$DATABASE ;
```

Try this query with your favourite interactive SQL client program!

Setting or resetting generator values

`SET GENERATOR` sets a starting value for a newly created generator, or resets the value of an existing generator. The new value for the generator, *int*, can be any integer in the range bounded by a `NUMERIC(18,0)` type, viz. -2^{31} to $2^{31}-1$.

Syntax for `SET GENERATOR`

```
SET GENERATOR NAME TO int;
```

Next time the `GEN_ID()` function is called, that value is *int* plus the increment specified in the `GEN_ID()` *step interval* parameter.

The following statement sets a generator value to 1,000:



Symbols **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

```
SET GENERATOR CUST_NO_GEN TO 1000;
```

When the GEN_ID() function is next called, that value returned will be *int* plus the increment specified in the GEN_ID() *step* parameter.

❖ Don't reset a generator unless you are certain that duplicate numbers will not occur. For example, generators are often used to assign a number to a column that has PRIMARY KEY or UNIQUE integrity constraints. If you reset such a generator so that it generates duplicates of existing column values, all subsequent insertions and updates fail with a "Key violation" error message. *Even if the reset series manages to avoid a key violation, thanks to gaps in the old sequence, there is still a risk that newly-generated keys will connect up with previously orphaned dependencies and cause undetected integrity violations.*

Deleting a generator

A generator which is no longer in use can be removed from the database using a DROP GENERATOR statement.

Syntax for DROP GENERATOR

```
DROP GENERATOR generator_name;
```

The command will fail if there are any triggers or procedures in the database which invoke that generator. It will be necessary first to find and drop the dependent code—or change your plans!



MORE INFORMATION

For further information about using generators in triggers and stored procedures, refer to the relevant topics in chapter 25, [Programming on Firebird Server](#).

[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

CHAPTER 16

Character Sets and Collation Orders

Firebird supports a broad variety of international character sets, including a number of two-byte sets and a three-byte Unicode set. Firebird provides choices of collation (sorting) order in many cases. This chapter lists the available character sets and their corresponding collation orders, and describes how to define

- the global default character set for a database
- alternative character set and collation order for a particular text column in a table
- the character set that the server should use when translating data between itself and the client
- collation order for
 - a text value in a comparison operation
 - an ORDER BY or GROUP BY clause

About...

Character sets

Specifying a default *character set* is intended primarily for users who need to provide a database for international use. A character set is a collection of symbols that includes at least one *character repertoire*. A character repertoire is a set of characters used by a particular culture for its publications, written communication and—in the context of a database—for computer input and output. For example, ISO Latin_1 is a character set that encompasses the English repertoire (A, B, C … Z) and the French repertoire (À, Á, Â, Ñ, Ç, Ð … Ù), making it useful for systems that span both cultural communities.

Naming of character sets

Most Firebird character sets are defined by standards. For example, the International Standards Organization defines ISO-8859-1, and Microsoft defines Windows1252. Firebird character set names follow the standard

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

that defines them; character set ISO8859_1 is "the set of characters defined by ISO Standard 8859-1, encoded by the values defined in ISO standard 8859-1, having each value represented by a single 8-bit byte".

Aliases

Character-set alias names support differences between platforms; for example, WIN1252 and WIN_1252.

Collation orders

Each character set has a default *collation order* that specifies how its symbols are sorted and ordered. Collation order determines the rules Firebird uses to sort, compare, and transliterate character data.

The default collation order implements *binary collation* for the character set: Binary collation sorts a character set by the numeric codes used to represent the characters. Some character sets support alternative collation orders. In each case, a particular choice of character set determines the choice of collation orders.

Naming of collation orders

Many Firebird collation names use the naming convention `xx_yy`, where `xx` is a two-letter language code, and `yy` is a two-letter country code. For example, `DE_DE` is the collation name for German as used in Germany; `FR_FR` is for French as used in France; and `FR_CA` is for French as used in Canada.

Firebird collations that emulate Paradox or dBASE collations have names like `PDOXXXXX`, `PXWXXX`, or `DBXXXX`. These names should match the names used in Paradox and dBASE.

 For more information about character sets and collation orders, download the Firebird Collation Kit provided by Brookstone Systems at <http://www.brookstonesystems.com>.

Default character set

If you do not specify a default character set, the character set defaults to `NONE` in most situations. Using character set `NONE` means that no character set assumptions are made for text columns; data are stored and retrieved exactly as entered.



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

Defaults per database element

Newly created database

The character set for a new database is NONE, unless you create the database using a DEFAULT CHARACTER SET specification.



Refer to chapter 14, [Specifying the default character set](#) for more information.

Connection

The character set for a connection is NONE, unless specified otherwise using

- SET NAMES in embedded applications or in `isql` (chapter 10, [Interactive SQL Utility \(isql\)](#), see [SET NAMES](#) on page 185)
- the `isc_dpb_lc_ctype` parameter of the database parameter block (DPB) for the `isc_attach_database()` function of the API



RAD database connection classes generally surface this parameter as a property

Statement

The character set for text values in a statement is interpreted according to be connection's character set *at run-time* (not according to the character set defined for the column when it was created) unless you specify a *character set marker* to indicate a different character set.

Character set marker

A character set marker consists of the character set name prefixed by an underscore character. For example, the character set marker for UNICODE_FSS is `_UNICODE_FSS`. Position the marker directly to the left of the text value being marked. For example,

```
INSERT INTO EMPLOYEE(Emp_ID, Emp_Name)
values(1234, _UNICODE_FSS 'Smith, John Joseph');
```

String literal

A string literal in a test or search condition, for example, in a WHERE clause, is, likewise, interpreted according to the connection's character set, *at the time the condition is tested*. Again, in these circumstances, a



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

character set marker will be required if the database column being searched has a character set which is different to that of the client connection:

```
... WHERE name = _ISO8859_1 'joe';
```

- ★ When you are developing with mixed character sets, it is good practice to use character set markers as a matter of form, especially if your application will cross the boundaries of multiple databases and/or it will be deployed internationally.

Character set storage requirements

Knowing the storage requirements of a particular character set is important, because in the case of CHAR columns, Firebird restricts the maximum amount of storage in each field in the column to 32,767 bytes (VARCHAR is restricted to 32,765 bytes).

- For character sets that require only a single byte of storage, the maximum number of symbols that can be stored in a single field corresponds to the number of bytes.
- For character sets that use two or three bytes per symbol, the maximum number of symbols that can be safely stored in a field is reduced to half or 1/3 of the maximum number of bytes for the datatype.

For example, for a CHAR column defined to use the UNICODE_FSS character set, the maximum number of characters that can be specified is 10,922 (32,767/3):

```
...  
CHAR(10922) CHARACTER SET UNICODE_FSS,  
...
```

Collation orders and index key size

If you specify a non-binary collation (one other than the default collation) for a character set, the index key can become larger than the stored string if the collation includes second-, third-, or fourth-order effects.

Collations for ISO8859_1, for example, use a full dictionary sort, with spaces and punctuation of fourth-order importance:



Symbols **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

First order:A is different from B

Second order:A is different from Å

Third order:A is different from a

Fourth order:The type of punctuation mark is important

Example Redwing

Red wing

Red-wing

Redwood

Red wood

Red worm

If spaces and punctuation marks are treated instead as a *first-order* difference, the same list would be sorted as follows:

Example Redwing

Redwood

Red wing

Red wood

Red worm

Red-wing

When an index is created, it uses the collation order defined for each text segment in the index. Because ISO8859_1 is a one-byte character set, if you use the default collation, the index key can hold about 253 characters (less, if it is a multi-segment key). However, if you choose an alternative collation for ISO8859_1, the index key can hold only 84 characters, even though the characters in the field being indexed each occupy only one byte.

- ❖ Some ISO8859_1 collations, DE_DE for example, require an average of three bytes per character for an index key.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Special character sets

Several character sets deserve special mention.

NONE, OCTETS, and ASCII

Each of these character sets represents a sequence of bytes, with the meanings shown in Table 16–1.

TABLE 16–1 Character sets NONE, OCTETS, and ASCII

Name	Description
NONE	Each byte is part of a character sequence, but there is no information about which character set it belongs to. NONE is provided for backward compatibility with versions of InterBase prior to Version 5. The user (for example, an application programmer) is responsible for character fidelity.
OCTETS	Bytes that do not represent characters.
ASCII	Values 0..127 are defined by ASCII; values outside that range are not characters, but are preserved. Firebird is fairly liberal about transliterating bytes in the 0..127 range of ASCII characters.
All others	Every byte has a specific definition, as per the standard for the character set.

ISO8859_1 (LATIN_1) and WIN1252

The Firebird ISO8859_1 character set is often specified to support European languages. ISO8859_1, also known as LATIN_1, is a proper subset of WIN1252: Microsoft added characters in positions that ISO specifically defines as *Not a character* (not “undefined”, but specifically “not a character”). Firebird supports both WIN1252 and ISO8859_1. You can always transliterate ISO8859_1 to WIN1252, but transliterating WIN1252 to ISO8859_1 can result in errors.



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

Support for Paradox and dBASE

Many character sets and their corresponding collations are provided to support Borland Paradox for DOS, Paradox for Windows, dBASE for DOS, and dBASE for Windows.

Character sets for DOS

The following character sets correspond to MS-DOS code pages, and should be used to specify character sets for Firebird databases that are accessed by Paradox for DOS and dBASE for DOS:

TABLE 16–2 Character sets corresponding to DOS code pages

Character set	DOS code page
DOS437	437
DOS850	850
DOS852	852
DOS857	857
DOS860	860
DOS861	861
DOS863	863
DOS865	865

The names of collation orders for these character sets that are specific to Paradox begin "PDOX". For example, the DOS865 character set for DOS code page 865 supports a Paradox collation order for Norwegian and Danish called "PDOX_NORDAN4".

The names of collation orders for these character sets that are specific to dBASE begin "DB". For example, the DOS437 character set for DOS code page 437 supports a dBASE collation order for Spanish called "DB_ESP437".



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Character sets for Microsoft Windows

There are five character sets that support Windows client applications, such as Paradox for Windows. These character sets are WIN1250, WIN1251, WIN1252, WIN1253, and WIN1254.

The names of collation orders for these character sets that are specific to Paradox for Windows begin "PXW". For example, the WIN1252 character set supports a Paradox for Windows collation order for Norwegian and Danish called "PXW_NORDAN4".

For more information about Windows character sets and Paradox for Windows collation orders, see the appropriate Paradox for Windows documentation and driver books.

Additional character sets and collations

Support for additional character sets and collation orders is constantly being added to Firebird. To see if additional character sets and collations are available for a newly created database, connect to the database with ISQL, then use the following set of queries to generate a list of available character sets and collations:

```
SELECT RDB$CHARACTER_SET_NAME, RDB$CHARACTER_SET_ID  
FROM RDB$CHARACTER_SETS  
ORDER BY RDB$CHARACTER_SET_NAME;  
  
SELECT RDB$COLLATION_NAME, RDB$CHARACTER_SET_ID  
FROM RDB$COLLATIONS  
ORDER BY RDB$COLLATION_NAME;
```

Transliteration

Converting characters in one Firebird character set to another character set, or to another SQL form of use, is *transliteration*. Converting from Unicode in the multi-byte FSS format to Unicode in wide-character format is an example of transliteration. Converting from DOS437 to ISO8859_1 is also transliteration.

Transliteration errors

Firebird reports a transliteration error if a character in the input set does not have an exact representation in the output set; Firebird does not transliterate a "missing character" to some other character. This restriction

**Symbols** [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

allows you to perform round-trip conversions: from character set A to character set B, then back to A, producing the original sequence of characters in the original representation.

Specifying character sets

This section provides details on how to specify character sets for databases and client attachments.

Default character set for a database

A database's default character set designation specifies the character set the server uses to tag CHAR, VARCHAR, and text blob columns in the database when no other character set information is provided. When data is stored in such columns without additional character set information, the server uses the tag to determine how to store and transliterate that data. A default character set should always be specified for a database when it is created with CREATE DATABASE.

To specify a default character set, use the DEFAULT CHARACTER SET clause of CREATE DATABASE. For example, the following statement creates a database that uses the ISO8859_1 character set:

```
CREATE DATABASE 'europe.gdb' DEFAULT CHARACTER SET ISO8859_1;
```

- ❖ If you do not specify a character set, the character set defaults to NONE. Using character set NONE means that there is no character set assumption for columns; data is stored and retrieved just as you originally entered it. You can load any character set into a column defined with NONE, but you cannot later move that data into another column that has been defined with a different character set. In this case, no transliteration is performed between the source and destination character sets, and errors may occur during assignment.



For the complete syntax of CREATE DATABASE, see [Using CREATE DATABASE](#) on p. 275 of chapter 14.

Character set for a column in a table

Character sets for individual columns in a table can be specified as part of the column's CHAR, VARCHAR or BLOB data type definition. When a character set is defined at the column level, it overrides the default character set declared for the database.

**Symbols** [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

For example, the following ISQL statements create a database with a default character set of ISO8859_1, then create a table where two column definitions include a different character set specification:

```
CREATE DATABASE 'europe.gdb' DEFAULT CHARACTER SET ISO8859_1;

CREATE TABLE RUS_NAME(
    LNAME VARCHAR(30) NOT NULL CHARACTER SET CYRL,
    FNAME VARCHAR(20) NOT NULL CHARACTER SET CYRL,
);
```

For the complete syntax of CREATE TABLE, see [Creating tables](#) on p. 315 of chapter 17.

Character set for a client attachment

When a client application, such as `isql`, connects to a database, it may have its own character set requirements. The server providing database access to the client does not know about these requirements unless the client specifies them. The client application specifies its character set requirement using the `SET NAMES` statement or an API function *before* it connects to the database.

`SET NAMES` and its equivalent in the API function call `isc_expand_db()` specify the character set the server should use when translating data from the database to the client application. Similarly, when the client sends data to the database, the server translates the data from the client's character set to the database's default character set (or the character set for an individual column if it differs from the database's default character set).

For example, the following ISQL command specifies that `isql` is using the DOS437 character set. The next command connects to the *europe* database:

```
SET NAMES DOS437;
CONNECT 'europe.gdb' USER 'JAMES' PASSWORD 'U4EEAH';
```



For more information about `SET NAMES`, refer to chapter 10, [Interactive SQL Utility \(isql\)](#), see [SET NAMES](#) on page 185. See the *Firebird Reference Guide* for details of the parameters for set names in embedded SQL.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

★ API applications set the character set in the *isc_dpb_lc_ctype* parameter of the database parameter block (DPB) for the *isc_attach_database()* function. RAD database connection classes generally surface this parameter as a property.

Specifying collation orders

This section describes how to specify collation orders for character sets in table columns, in comparisons, and in ORDER BY and GROUP BY clauses.

Collation order for a column

When a CHAR or VARCHAR column is created for a table, either with CREATE TABLE or ALTER TABLE, the collation order for the column can be specified using the COLLATE clause. COLLATE is especially useful for character sets such as ISO8859_1 or DOS437 that support many different collation orders.

For example, the following ISQL ALTER TABLE statement adds a new column to a table, and specifies both a character set and a collation order:

```
ALTER TABLE 'FR_CA_EMP'  
ADD ADDRESS VARCHAR(40) CHARACTER SET ISO8859_1 NOT NULL  
COLLATE FR_CA;
```

For the complete syntax of ALTER TABLE, see [Altering tables](#) on p. 340 of chapter 17.

Collation order in comparisons

When CHAR or VARCHAR values are compared in a WHERE clause, it can be necessary to specify a collation order for the comparisons if the values being compared use different collation orders.

To specify the collation order to use for a value during a comparison, include a COLLATE clause after the value. For example, in the following WHERE clause fragment from an embedded application, the value to the left of the comparison operator is forced to be compared using a specific collation:

```
WHERE LNAME COLLATE FR_CA = :lname_search;
```

For the complete syntax of the WHERE clause, see the *Firebird Reference Guide*—[SELECT](#) (ch. 2 p. 191).



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Collation order in ORDER BY

When CHAR or VARCHAR columns are ordered in a SELECT statement, it can be necessary to specify a collation order for the ordering, especially if columns used for ordering use different collation orders.

To specify the collation order to use for ordering a column in the ORDER BY clause, include a COLLATE clause after the column name. For example, in the following ORDER BY clause, the collation order for two columns is specified:

```
    . . .
ORDER BY LNAME COLLATE FR_CA, FNAME COLLATE FR_CA;
```

For the complete syntax of the ORDER BY clause, see the *Firebird Reference Guide*.

Collation order in a GROUP BY clause

When CHAR or VARCHAR columns are grouped in a SELECT statement, it can be necessary to specify a collation order for the grouping, especially if columns used for grouping use different collation orders.

To specify the collation order to use for grouping columns in the GROUP BY clause, include a COLLATE clause after the column name. For example, in the following GROUP BY clause, the collation order for two columns is specified:

```
    . . .
GROUP BY LNAME COLLATE FR_CA, FNAME COLLATE FR_CA;
```

For the complete syntax of the GROUP BY clause, see the *Firebird Reference Guide*—[SELECT](#) (ch. 2 p. 191).

Also see [The GROUP BY clause](#) on p. 135 of chapter 9.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

CHAPTER 17

Tables

In SQL-89 and SQL-92 terms Firebird tables are *persistent base tables*. The standards define several other types, including *viewed tables*, which Firebird implements as views (see chapter 19, [Views](#) (p. 363)) and *derived tables*, which Firebird could be said to implement as selectable stored procedures (see chapter 25, [Error trapping and handling](#) (p. 549)).

This chapter examines Firebird's persistent base tables and their elements and relationships—how to create and modify them and how to dispose of them when they are no longer needed.

About Firebird tables

Physical considerations

Unlike desktop databases, such as Paradox and xBase databases, a Firebird database is not a series of "table files" physically organized in rows and columns. Firebird stores data, independently of its structure, in a compressed format, on *database pages*. It may store one or many records—or, correctly, *rows*—of a table's data on a single page. In cases where the data for one row are too large to fit on one page, a row may span multiple pages.

Although a page which stores table data will always contain only data belonging to one table, pages are not stored contiguously. The data for a table are scattered all around the disk and, in multi-file databases, may be dispersed across several directories or disks. BLOB data are stored apart from the rows that own them, in another style of database page.

Structural descriptions

Metadata—the physical descriptions of tables and their columns and attributes, as well as those of all other objects—are themselves stored in ordinary Firebird tables inside the database. The Firebird engine writes to

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

these tables when database objects are created, modified or destroyed. It refers to them constantly when carrying out operations on rows. These tables are known as *system tables*.

System tables

All of the tables which store the descriptions of database objects have identifiers beginning with "RDB\$". For example, the table RDB\$RELATIONS store all of the persistent table descriptions. This "database within a database" is highly normalized. It is possible to alter the data in the system tables by performing regular SQL operations on them. Some desktop tools, such as **isql** and **gfix**, do change data in the system tables.

- ❖ It is NOT recommended that you attempt to bypass DDL by altering the system tables yourself, either through application code or by way of an interactive query tool. DDL statements are designed to perform operations on the metadata tables safely and in full cognisance of the cascading effects. As a sophisticated database management system, Firebird was not designed with raw end-user queries on the system tables in mind.

For details of these tables, see *Firebird Reference Guide*— [System Tables and Views](#) (ch. 9 p. 346).

Preparing to create tables

It is assumed that, having reached the point where you are ready to create tables, you have already prepared your data analysis, modeling and design (see chapter 11, [Designing Databases](#) (p. 199)) and have very clear blueprint for the structures of your main tables and their relationships. In preparation for creating these tables, you need to have performed these steps:

- You have created a database to accommodate them.
 For instructions, see [Creating a database](#) on page 274 in chapter 14.
- You have connected to the database—see [Connecting to a database](#) on page 282 in chapter 14.
- If you plan to use domains for the data type definitions of your tables' columns, you have already created the domains in advance of starting to create tables—see chapter 15, [Firebird domains](#) (p. 285).



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Table ownership

When a table is created, Firebird automatically applies the default SQL security scheme to it. The person who creates the table (the owner), is assigned all privileges for it, including the right to grant privileges to other users, triggers, and stored procedures.



For more information on security, see [Database-level security](#) on p. 429 of chapter 22.

Creating tables

You can create tables in the database with the CREATE TABLE statement. The syntax for CREATE TABLE is:

```
CREATE TABLE table [EXTERNAL [FILE] 'filespec' ]  
    (<col_def> [, <col_def> | <tconstraint> ...]);
```

The first argument that you supply to CREATE TABLE is the table name, which is required, and must be unique among all table and procedure names in the database. You must also supply at least one column definition.

★ If you are coding CREATE TABLE in an embedded SQL application, with the intention of also populating the data for this table from the application, the table must first be *declared* using DECLARE TABLE. The DECLARE TABLE statement must precede CREATE TABLE.



Detailed syntax specifications:

- [DECLARE TABLE](#)
- [CREATE TABLE](#) (*Firebird Reference Guide—SQL Statement and Function Reference* (ch. 2 p. 43))

Defining columns

When you create a table in the database, your main task is to define the various attributes and constraints for each of the columns in the table. The syntax for defining a column is:

```
<col_def> = col {datatype | COMPUTED [BY] (<expr>) | domain}  
    [DEFAULT {literal | NULL | USER}]  
    [NOT NULL] [<col_constraint>]  
    [COLLATE collation]
```

The next sections list the required and optional attributes that you can define for a column.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Required attributes

You are required to specify:

- A column name, which must be unique among the columns in the table.
- One of the following:
 - A SQL datatype (*datatype*).
 - An expression (*expr*) for a computed column.
 - A domain definition (*domain*) for a domain-based column.

Optional attributes

You can optionally specify:

- A *default value* for the column.
- *Integrity constraints*. Constraints can be applied to a set of one or more columns (a table-level constraint), or to a single column (a column-level constraint). Integrity constraints include:
 - The PRIMARY KEY column constraint, if the column is a PRIMARY KEY, and the PRIMARY KEY constraint is not defined at the table level.
 - ❖ Creating a PRIMARY KEY requires exclusive database access.
 - The UNIQUE constraint, if the column is not a PRIMARY KEY, but still needs to disallow duplicate and NULL values.
 - The FOREIGN KEY constraint, if the column references a PRIMARY KEY in another table. The foreign key constraint includes the optional ON UPDATE and ON DELETE mechanisms for specifying what happens to the foreign key when the primary key is updated (cascading referential integrity).
 - ❖ Creating a FOREIGN KEY requires exclusive database access.
- A *NOT NULL attribute* if the column does not allow NULL values and there is no NOT NULL attribute inherited from a domain.
- ❖ This attribute is required if the column is a PRIMARY KEY or UNIQUE key.
- ❖ Firebird does not support a *NULABLE* attribute. In compliance with standards, all columns in Firebird are nullable unless explicitly constrained to be NOT NULL. Do not use a domain with a NOT NULL constraint when defining a column that is allowed to be NULL.
- A *CHECK constraint* for the column. A CHECK constraint enforces a condition that must be true in order for an insert or an update to a column or group of columns to be allowed.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- A *CHARACTER SET* can be specified for an individual character or text BLOB column when you define the datatype. If you do not specify a character set, the column assumes the database character set as a default.
- A collation order (*COLLATE*) for an individual character column. Collation order is not valid for BLOB types.

Specifying the datatype

When creating a table, you must specify the data type for each column as the first attribute following the column identifier. The data type predetermines

- the set of valid data that the column can contain
- the set of allowable operations that can be performed on the data
- the disk space requirements for each data item.

Syntax

```
<datatype> =
{SMALLINT | INTEGER | FLOAT | DOUBLE PRECISION} [<array_dim>]
| {DATE | TIME | TIMESTAMP} [<array_dim>]
| {DECIMAL | NUMERIC} [(precision [, scale])] [<array_dim>]
| {CHAR | CHARACTER | CHARACTER VARYING | VARCHAR} [(int)]
[<array_dim>] [CHARACTER SET charname]
| {NCHAR | NATIONAL CHARACTER | NATIONAL CHAR}
[VARYING] [(int)] [<array_dim>]
| BLOB [SUB_TYPE {int | subtype_name}] [SEGMENT SIZE int]
[CHARACTER SET charname]
| BLOB [(seglen [, subtype])]

<array_dim> = [x:y [, x1:y1 ...]]
```

❖ The outermost (boldface) brackets must be included when declaring arrays.



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

Supported datatypes

The general categories of datatypes that are supported include:

- Character datatypes.
- Integer datatypes.
- Decimal datatypes, both fixed and floating.
- A DATE datatype to represent the date, a TIME datatype to represent the time, and a TIMESTAMP datatype to represent both the date and time.
- A BLOB datatype to represent unstructured binary data, such as graphics and digitized voice.
- Arrays of datatypes (except for BLOB data).

 For a complete list and description of data types that Firebird supports, see [Supported Datatypes](#) in the Firebird Reference Guide. For a detailed discussion, see chapter 13, [Firebird Data Types](#) (p. 231).

Defining a character set

The datatype specification for a CHAR, VARCHAR, or BLOB text column definition can include a CHARACTER SET clause to specify a particular character set for a column. If you do not specify a character set, the column assumes the default database character set. If the database default character set is subsequently changed, all columns defined after the change have the new character set, but existing columns are not affected. For detailed information, see chapter 16, [Character Sets and Collation Orders](#) (p. 301). For a list of available character sets recognized by Firebird, refer to the *Firebird Reference Guide*— [Character Sets and Collation Orders](#) (ch. 4 p. 249).

The COLLATE clause

The COLLATE clause allows you to specify a particular collation order (sort order) for columns defined as CHAR and VARCHAR text datatypes. You must choose a collation order that is supported for the column's character set. If the column does not inherit a character set from a domain and does not specify one in its definition, then the character set is that which is the default character set for the database.

- ❖ If collation order is set at the column level, it overrides any collation order set at the domain level.
- ❖ When non-binary collations are used, the index key can become larger than the stored string if the collation includes secondary- or further-order effects.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

In the following statement, BOOKNO keeps the default collating order for the database's default character set.

The second (TITLE) and third (EUROPUB) columns specify different character sets and collating orders.

```
CREATE TABLE BOOKADVANCE (BOOKNO CHAR(6),  
    TITLE CHAR(50) CHARACTER SET DOS437 COLLATE PDOX_INTL,  
    EUROPUB CHAR(50) CHARACTER SET ISO8859_1 COLLATE FR_FR);
```

For a list of the available characters sets and collation orders that Firebird recognizes, see the *Firebird Reference Guide*.

Defining domain-based columns

A *domain* is a package of column attributes that is predefined for global use, providing a consistent data format for similar items that are being stored in different tables. Domains must be created with the CREATE DOMAIN statement before you can reference them to define columns locally. When you create a table, you can then substitute the mandatory data type parameter with a domain name to have the new column inherit the package of attributes defined for the domain.

 For information on how to create a domain, see [Using CREATE DOMAIN](#) on page 286 in chapter 15.

The column definition can include a new default value, additional CHECK constraints, or a collation clause that overrides the domain definition. It can also include additional column constraints. For example, you can specify a NOT NULL setting if the domain does not already define one.

❖ You cannot override the domain's NOT NULL setting with a local column definition.

For example, the following statement creates a table, COUNTRY, referencing the domain, COUNTRYNAME, which was previously defined with a datatype of VARCHAR(15):

```
CREATE TABLE COUNTRY  
(COUNTRY COUNTRYNAME NOT NULL PRIMARY KEY,  
CURRENCY VARCHAR(10) NOT NULL);
```



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Defining COMPUTED (expression-based) columns

A computed column is one whose value is calculated each time the column is accessed at run time. The syntax is:

```
<col_name> COMPUTED [BY] (<expr>);
```

If you do not specify the datatype, Firebird calculates an appropriate one. *expr* is any arithmetic expression that is valid for the datatypes in the columns; it must return a single value, and cannot be an array or return an array. Columns referenced in the expression must exist before the COMPUTED [BY] clause can be defined.

Constraints

Constraints are not applicable to computed columns.

- ★ Firebird does not return an error if you define a constraint on a computed column, but constraints declared on computed columns will not be honored.

Examples of COMPUTED columns

The following statement creates a computed column, FULL_NAME, by concatenating the LAST_NAME and FIRST_NAME columns.

```
CREATE TABLE EMPLOYEE  
  (FIRST_NAME VARCHAR(10) NOT NULL,  
   LAST_NAME VARCHAR(15) NOT NULL,  
   FULL_NAME COMPUTED BY (LAST_NAME || ' ' || FIRST_NAME));
```

The next statement computes two columns using context variables. This can be useful for logging the particulars of row creation:

```
CREATE TABLE SNIFFIT  
  (SNIFFID INTEGER NOT NULL,  
   SNIFF COMPUTED BY (CURRENT_USER),  
   SNIFFDATE COMPUTED BY (CURRENT_TIMESTAMP));
```

The next example creates a table with a calculated column (NEW_SALARY) using the previously created EMPNO and SALARY domains.

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

```
CREATE TABLE SALARY_HISTORY (EMP_NO EMPNO NOT NULL,
CHANGE_DATE DATE DEFAULT 'NOW' NOT NULL,
UPDATER_ID VARCHAR(20) NOT NULL,
OLD_SALARY SALARY NOT NULL,
PERCENT_CHANGE DOUBLE PRECISION
DEFAULT 0
NOT NULL
CHECK (PERCENT_CHANGE BETWEEN -50 AND 50),
NEW_SALARY COMPUTED BY
(OLD_SALARY + OLD_SALARY * PERCENT_CHANGE / 100),
PRIMARY KEY (EMP_NO, CHANGE_DATE, UPDATER_ID),
FOREIGN KEY (EMP_NO) REFERENCES EMPLOYEE (EMP_NO)
ON UPDATE CASCADE
ON DELETE CASCADE);
```

Specifying column default values

You can optionally define a default value that is automatically written into a column if a row is inserted without including the column in the insert statements column list. Defaults can save data entry time and prevent data entry errors.

- ★ Defaults defined at the column level with CREATE TABLE or ALTER TABLE override defaults inherited from a domain.

For example, a possible default for a DATE column could be the context variable CURRENT_DATE (today's date) or, in a (Y/N) flag column for saving changes, 'Y' could be the default.

Default values can be:

- *a constant*: The default value is a user-specified string, numeric value, or date value.
- *NULL*: If the user does not enter a value, a NULL value is entered into the column.
- a Firebird *predefined date literal*
- *USER, CURRENT_USER or CURRENT_ROLE*: The default is the name of the current user or role.

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

- ❖ If your operating system supports the use of multi-byte characters in user names, or you have used a multi-byte character set when defining roles, then any column into which these defaults will be stored must be defined using a compatible character set.
- *Other context variables* (CURRENT_DATE, CURRENT_TIMESTAMP, CURRENT_TIME)

When defaults won't work

It is a common mistake to assume that a default value will be used whenever Firebird receives NULL in a defaulted column. When relying on defaults, it must be understood that a default will be applied

- only upon insertion of a new row

AND

- only if the INSERT statement does not include the defaulted column in its column list

- ❖ If your application includes the defaulted column in the INSERT statement and sends NULL in the values list, then NULL will be stored, regardless of any default defined.

In the following example, the first statement creates a domain with USER named as the default. The next statement creates a table that includes a column, ENTERED_BY, based on the USERNAME domain.

```
CREATE DOMAIN USERNAME AS VARCHAR(20)
DEFAULT USER;

CREATE TABLE ORDERS (ORDER_DATE DATE, ENTERED_BY USERNAME,
ORDER_AMT DECIMAL(8,2));
INSERT INTO ORDERS (ORDER_DATE, ORDER_AMT)
VALUES ('1-MAY-93', 512.36);
```

The INSERT statement does not include a value for the ENTERED_BY column, so Firebird automatically inserts the user name of the current user, JSMITH:

```
SELECT * FROM ORDERS;
```

Specifying NOT NULL

You can optionally specify NOT NULL to force the user to enter a value. If you do *not* specify NOT NULL, then NULL values are allowed in the column.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- ❖ You cannot override a NOT NULL setting that has been set at a domain level with a local column definition.

If you have already specified NULL as a default value, be sure not to create contradictory constraints by also specifying the NOT NULL attribute, as in the following example:

```
CREATE TABLE MY_TABLE (COUNT INTEGER DEFAULT NULL NOT NULL);
```

Defining integrity constraints

Firebird allows you to optionally apply certain constraints to a column, called *integrity constraints*, which are the rules that govern column-to-table and table-to-table relationships, and validate data entries. They span all transactions that access the database and are automatically maintained by the system. Integrity constraints can be applied to an entire table or to an individual column.

PRIMARY KEY and UNIQUE constraints

The PRIMARY KEY and UNIQUE integrity constraints ensure that the values entered into a column or set of columns are unique in each row. If you try to insert a duplicate value in a PRIMARY KEY or UNIQUE column, Firebird returns an error. When you define a UNIQUE or PRIMARY KEY column, determine whether the data stored in the column are inherently unique. To take an example, no two social security numbers or driver's license numbers are ever the same, so you might place a UNIQUE constraint on either of these columns and expect that any insert or update attempt that caused a violation contained suspect data.

In the EMPLOYEE table below, EMP_NO is the primary key that uniquely identifies each employee. EMP_NO is the primary key because no two values in the column are alike. . .

TABLE 17-1 The EMPLOYEE table

EMP_NO	LAST_NAME	FIRST_NAME	JOB_TITLE	PHONE_EXT
10335	Smith	John	Engineer	4968
21347	Carter	Catherine	Product Manager	4967
13314	Jones	Sarah	Senior Writer	4800



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

If the EMP_NO column did not exist, then no other column is a candidate for primary key due to the high probability for duplication of values.

- LAST_NAME, FIRST_NAME, and JOB_TITLE fail because more than one employee can have the same first name, last name, and job title.
- A primary key that combines LAST_NAME and PHONE_EXT might work, but there could be two people with identical last names at the same extension.
- In this table, the EMP_NO column is actually the only acceptable candidate for the primary key because it guarantees a unique number for each employee in the table.
- However, even EMP_NO may be considered suspect because it may not be completely *atomic*.

Atomicity of PRIMARY KEY columns

It is recommended practice to avoid placing the PRIMARY KEY constraint on meaningful data such as driver's licence or telephone number—even if they are systematically unique—because all external data must be considered vulnerable to error or change. A *surrogate primary key* is completely reliable and *atomic*.

A **surrogate primary key** is one that substitutes for (stands in place of) one or more columns that, when isolated, uniquely identify a row. The surrogated column or columns instead take a UNIQUE constraint, without themselves being constrained as PRIMARY KEY.

Atomic is the word that describes a data item which has no meaning at all as data. Its sole work is structural.

Firebird provides an excellent mechanism for creating fully atomic surrogate PRIMARY KEY columns: generators.



For more information, see [Firebird Generators](#) on page 296 in chapter 15.

Composite primary keys

During data analysis, it sometimes happens that no single unique column can be found in the data structure. Theoretical wisdom suggests that the next best thing is to take two or more columns that, when grouped together, will ensure a unique row. When multiple columns are conjoined to form a primary, the key is called a *composite key*. Composite keys, even if all elements are atomic, should be used with great restraint, because of the following limitations:

- any FOREIGN KEYS in other tables that reference this table will have to propagate every element of the composite key. With non-atomic key elements, the referential integrity of the relationship is made



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

vulnerable to external changes and errors. Even if all elements are atomic, it still forces referential links to be more complex than necessary.

- keys have mandatory indexes created for them. Composite indexes have stricter size limits than single-column indexes.
- composite indexes tend to be large. Large indexes use more database pages, causing indexed operations (sorts, joins, comparisons) to be slower than is necessary.

Consider the benefits of using a surrogate primary key wherever your only candidate key is composite.

Declaring the PRIMARY KEY

As a column attribute

Firebird allows you to define a table's PRIMARY KEY as a *column constraint*, for example,

```
CREATE TABLE ATABLE (ID D_IDENTITY NOT NULL PRIMARY KEY,  
...);
```

As a named table constraint

Alternatively, you can define a *named table-level constraint* for the PRIMARY KEY, for example,

```
CREATE TABLE ATABLE (ID D_IDENTITY NOT NULL,  
...<other columns>,)  
CONSTRAINT PK_ATABLE PRIMARY KEY(ID));
```



For more information, see [Named constraints](#) on page 333.

A table can have only one primary key, so you will get an error if you attempt to define a primary key both ways. Either way, the NOT NULL attribute must be declared for *all* columns in a group of one or more columns which will be used as the primary key, in order to enforce the uniqueness requirement.



Many developers prefer to use the *named constraint* style for defining both primary and foreign keys. Apart from its benefits for self-documentation, this style enables you to separate the key definitions from the column definitions in scripts. The technique used is to define tables without keys and to add them later, perhaps in a separate script, using ALTER TABLE.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

The UNIQUE constraint

Because a unique constraint, like a primary key, ensures that no two rows have the same value for a specified column or group of columns, the NOT NULL attribute must be applied to all of the columns on which the UNIQUE constraint will operate.

You can have more than one UNIQUE constraint defined for a table, but it cannot be applied to the same set of columns that is used for either the PRIMARY KEY or another UNIQUE constraint.

UNIQUE keys

A column or column group that is constrained by UNIQUE can be referenced by a FOREIGN KEY constraint in another table.

The FOREIGN KEY constraint

A foreign key is a column or set of columns in one table that corresponds in exact order to a column or set of columns defined as a PRIMARY KEY or as a UNIQUE constraint in another table. The primary reason for defining foreign keys is to preserve the integrity of a relationship between two tables—*referential integrity*—by ensuring that

- a row in the referencing table cannot exist unless there is a unique corresponding row in the referenced table
- without modification of the FOREIGN KEY constraint by an ON DELETE action, a row in the referenced table cannot be deleted whilst there are rows in the referencing table which correspond to it

For example, in the following two tables, PROJECT.TEAM_LEADER is a foreign key referencing the primary key, EMP_NO in the EMPLOYEE table.

TABLE 17-2 The PROJECT table

PROJ_ID	TEAM_LEADER	PROJ_NAME	PROJ_DESC	PRODUCT
DGPII	44	Automap	blob data	hardware
VBASE	47	Video database	blob data	software
HWRII	24	Translator upgrade	blob data	software



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

TABLE 17-3 The EMPLOYEE table

EMP_NO	LAST_NAME	FIRST_NAME	DEPT_NO	JOB_CODE	PHONE_EXT	SALARY
24	Smith	John	100	Eng	4968	64000
48	Carter	Catherine	900	Sales	4967	72500
36	Smith	Jane	600	Admin	4800	37500

Referential integrity

Firebird enforces *referential integrity* in the following ways:

- The UNIQUE or PRIMARY KEY constraint on the referenced column or column group must already be defined and committed before a FOREIGN KEY constraint is permitted to refer to it
- A choice of *referential integrity behaviors* is provided by way of optional ON UPDATE and ON DELETE trigger action parameters for the REFERENCES sub-clause of the constraint or column definition.

Optional referential trigger actions

The following trigger action options are available when specifying the optional ON UPDATE and ON DELETE parameters for the REFERENCES clause of a FOREIGN KEY constraint::

TABLE 17-4 Referential integrity trigger options

Action specified	Effect on foreign key
NO ACTION	[Default] The foreign key does not change (can cause the primary key update or delete to fail due to referential integrity checks)
CASCADE	ON UPDATE: the foreign key will be updated to match the new value of the corresponding primary key or unique key ON DELETE: the row will be deleted when the row containing the corresponding primary key or unique key is deleted



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

TABLE 17–4 Referential integrity trigger options

Action specified	Effect on foreign key
SET DEFAULT	<p>The foreign key column(s) will be set to their DEFAULT values</p> <ul style="list-style-type: none">ON UPDATE: when the value in one or more of the columns of the corresponding primary key or unique key changesON DELETE: when the referenced row is deleted <p>❖ This trigger will fail if the default values for the referencing columns do not exist as values in the primary key column(s).</p> <ul style="list-style-type: none">The default value used is the one which is in effect when the referential integrity constraint was defined. If the default for a foreign key column gets changed later, the change does not propagate into the referential integrity constraint.
SET NULL	<p>Every column of the foreign key is set to NULL</p> <ul style="list-style-type: none">ON UPDATE: when the value in one or more of the columns of the corresponding primary key or unique key changesON DELETE: when the referenced row is deleted <p>❖ This trigger will fail if any of the referencing columns has the NOT NULL attribute.</p> <p>This action has a side-effect which may be undesirable—see Orphan rows on page 330.</p>

- If you do not use the ON UPDATE and ON DELETE options when defining foreign keys, you must make sure that your own triggers or your application code will preserve referential integrity when data in any key change. Triggers are much safer than application code, since they position the data integrity rules in the database.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

★ Using the previous pair of example tables, if a value in the EMP_NO column of the EMPLOYEE table (the primary key) is allowed to be changed, you would need to write an AFTER DELETE trigger that updates the TEAM_LEADER column of the PROJECT table (the foreign key).

- If you delete a row from the referenced table, you must first delete all rows that refer to it through foreign keys. The ON DELETE CASCADE trigger would perform this for you automatically.
- You cannot insert into or change a value in a column defined as a foreign key unless that value exists in the referenced primary key.

For example, to enter a value in the TEAM_LEADER column of the PROJECT table, that value must first exist in the EMP_NO column of the EMPLOYEE table.

The following example specifies that when a value is deleted from a primary key, the corresponding values in the foreign key are set to NULL. When the primary key is updated, the changes are cascaded so that the corresponding foreign key values match the new primary key values.

```
CREATE TABLE PROJECT {  
    . . .  
    TEAM LEADER INTEGER REFERENCES EMPLOYEE (EMP_NO)  
    ON DELETE SET NULL  
    ON UPDATE CASCADE  
    . . .};
```

Referring to tables owned by others

If you want to create a foreign key that refers to a table owned by someone else, that owner must first use the GRANT command to grant you REFERENCES privileges on that table. Alternately, the owner can grant REFERENCES privileges to a role and then grant that role to you.

For more information on granting privileges to users and roles, see [Database-level security](#) on p. 429 of chapter 22. See the *Firebird Reference Guide*—[SQL Statement and Function Reference](#) (ch. 2 p. 43) for full syntax.



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

Orphan rows

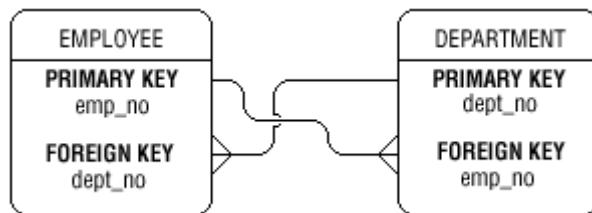
If foreign key columns are created without the NOT NULL constraint, it is possible to store *orphan rows* in the table. These are rows with NULL in the foreign key column(s). Without any link to the “parent” (the table referenced by the foreign key), such rows move outside the domain of the referential integrity relationship.

It is also possible for rows to become orphans when the SET NULL parameter is used as the trigger action for ON DELETE or ON UPDATE.

Circular references

A circular reference exists when two tables each refers the other's foreign keys *and* primary keys. In the following illustration, the foreign key in the EMPLOYEE table, DEPT_NO, references the primary key, DEPT_NO, in the DEPARTMENT table. Therefore, the primary key, DEPT_NO must be defined in the DEPARTMENT table before it can be referenced by a foreign key in the EMPLOYEE table. In the same manner, EMP_NO, which is the EMPLOYEE table's primary key, must be created before the DEPARTMENT table can define EMP_NO as its foreign key.

FIGURE 5 Circular references



Problems occur when you try to insert a new row into either table.

- Inserting a new row into the EMPLOYEE table causes a new value to be inserted into the DEPT_NO (foreign key) column, but you cannot insert a value into the foreign key column unless that value already exists in the DEPT_NO (primary key) column of the DEPARTMENT table.
- It is also true that you cannot add a new row to the DEPARTMENT table unless the values placed in the EMP_NO (foreign key) column already exist in the EMP_NO (primary key) column of the EMPLOYEE table. Therefore, you are in a deadlock situation because you cannot add a new row to either table!



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Firebird gets around the problem of circular referencing by allowing you to insert a NULL into a foreign key column—in effect, a foreign key constraint is not violated unless a *value* is inserted for which no referenced primary or unique key exists. NULL is not a value and causes no violation.

The following example illustrates the sequence for inserting a new row into each table:

- Insert a new row into the EMPLOYEE table by placing “1” in the EMP_NO primary key column, and a NULL in the DEPT_NO foreign key column.
- Insert a new row into the DEPARTMENT table, placing “2” in the DEPT_NO primary key column, and “1” in the foreign key column.
- Use ALTER TABLE to modify the EMPLOYEE table. Change the DEPT_NO column from NULL to “2.”

Defining a CHECK constraint

A CHECK constraint applies a condition or requirement on the column that an incoming data value must meet in order for the insert or update to succeed. The search condition can

- verify that the value entered falls within a certain permissible range
- compare the value for a match to one value in a list of values
- compare the value with a constant, an expression or with data values in other columns of the same row

❖ A CHECK constraint guarantees data integrity only when the values being verified are *in the same row* as the value being checked. Do not attempt to use expressions that compare the value with values in different rows of the same table or in different tables, since any row other than the current one is potentially in the process of being altered or deleted by another transaction, thus dooming “successful” constraint check to be invalid.

In the following example, the CHECK constraint is guaranteed to be satisfied:

```
CHECK (VALUE (COL_1 > COL_2));
INSERT INTO TABLE_1 (COL_1, COL_2) VALUES (5,6);
```



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Syntax

```
CHECK (<search condition>);

<search_condition> =
  <val> <operator> {<val> | (<select_one>)}
  | <val> [NOT] BETWEEN <val> AND <val>
  | <val> [NOT] LIKE <val> [ESCAPE <val>]
  | <val> [NOT] IN (<val> [, <val> ...] | <select_list>)
  | <val> IS [NOT] NULL
  | <val> {[NOT] {= | < | >} | >= | <=}
    {ALL | SOME | ANY} (<select_list>)
  | EXISTS (<select_expr>)
  | SINGULAR (<select_expr>)
  | <val> [NOT] CONTAINING <val>
  | <val> [NOT] STARTING [WITH] <val>
  | (<search_condition>)
  | NOT <search_condition>
  | <search_condition> OR <search_condition>
  | <search_condition> AND <search_condition>
```

Restrictions

- A CHECK constraint cannot refer to a domain.
- A column can have only one CHECK constraint, although its logic can be complex.
- On a domain-based column, you cannot override a CHECK constraint inherited from the domain. A column based on a domain can add *additional* CHECK constraints to the local column definition which will be ANDed to the inherited constraint.

In the next example, a CHECK constraint is placed on the SALARY domain. VALUE is a placeholder for the name of a column that will eventually be based on the domain.

```
CREATE DOMAIN BUDGET
  AS NUMERIC(12,2)
  DEFAULT 0
  CHECK (VALUE > 0);
```

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

The next statement illustrates PRIMARY KEY, FOREIGN KEY, CHECK, and the referential integrity constraints ON UPDATE and ON DELETE. The PRIMARY KEY constraint is based on three columns, so it is a table-level constraint. The FOREIGN KEY column (JOB_COUNTRY) references the PRIMARY KEY column (COUNTRY) in the table, COUNTRY. When the primary key changes, the ON UPDATE and ON DELETE clauses guarantee that the foreign key column will reflect the changes. This example also illustrates using domains (JOBCODE, JOBGRADE, COUNTRYNAME, SALARY) and a CHECK constraint to define columns:

```
CREATE TABLE JOB
  (JOB_CODE JOBCODE NOT NULL,
   JOB_GRADE JOBGRADE NOT NULL,
   JOB_COUNTRY COUNTRYNAME NOT NULL,
   JOB_TITLE VARCHAR(25) NOT NULL,
   MIN_SALARY SALARY NOT NULL,
   MAX_SALARY SALARY NOT NULL,
   JOB_REQUIREMENT BLOB(400,1),
   LANGUAGE_REQ VARCHAR(15) [5],
   PRIMARY KEY (JOB_CODE, JOB_GRADE, JOB_COUNTRY),
   FOREIGN KEY (JOB_COUNTRY) REFERENCES COUNTRY (COUNTRY)
   ON UPDATE CASCADE
   ON DELETE CASCADE,
   CHECK (MIN_SALARY < MAX_SALARY));
```

Named constraints

When declaring a table-level or a column-level constraint, you can optionally name the constraint using the CONSTRAINT clause. If you omit the CONSTRAINT clause, Firebird generates a unique system constraint name which is stored in the system table, RDB\$RELATION_CONSTRAINTS.

Although naming a constraint is optional, assigning a descriptive name with the CONSTRAINT clause can make the constraint easier to find for changing or dropping or when its name appears in a constraint violation error message.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

★ To ensure that the constraint names are visible in RDB\$RELATION_CONSTRAINTS, commit your transaction before trying to view the constraint in the RDB\$RELATION_CONSTRAINTS system table.

Syntax

The syntax for a column-level constraint is:

```
<col_constraint> = [CONSTRAINT constraint] <constraint_def>
[<col_constraint> ...]

<constraint_def> =
UNIQUE | PRIMARY KEY
| CHECK (<search_condition>)
| REFERENCES other_table [(other_col [, other_col ...]) ]
[ON DELETE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
[ON UPDATE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
```

The syntax for a table-level constraint is:

```
<tconstraint> = [CONSTRAINT constraint] <tconstraint_def>
[<tconstraint> ...]

<tconstraint_def> = {PRIMARY KEY | UNIQUE} (col [, col ...])
| FOREIGN KEY (col [, col ...])
REFERENCES other_table [(other_col [, other_col ...]) ]
[ON DELETE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
[ON UPDATE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
| CHECK (<search_condition>)
```

The following statement illustrates how to create a simple, column-level PRIMARY KEY constraint:

```
CREATE TABLE COUNTRY
(COUNTRY COUNTRYNAME NOT NULL PRIMARY KEY,
CURRENCY VARCHAR(10) NOT NULL);
```

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

The next example illustrates how to create a UNIQUE constraint at both the column level and the table level:

```
CREATE TABLE STOCK
(MODEL SMALLINT NOT NULL UNIQUE,
MODELNAME CHAR(10) NOT NULL,
ITEMID INTEGER NOT NULL,
CONSTRAINT MOD_UNIQUE UNIQUE (MODELNAME, ITEMID));
```

Using external files as tables

External files are filesystem-format ASCII text files that can be read and manipulated by non-Firebird applications. The EXTERNAL FILE keyword enables you to define a table whose row structure maps to fixed length "fields" in "records" (usually delimited by line-feeds) that reside in an external file. Firebird can SELECT from and INSERT into such a file as if it were a regular table.

- Firebird cannot UPDATE or DELETE records in an external file
- Shared access by Firebird and other applications at file level is not possible. However, the file can be modified by other applications at times when it is not opened by a Firebird transaction.
- The text file containing the data must be on a storage device that is physically under the control of the server.
- You can configure a list of directories where Firebird will search for external files—see [external_file_directory](#) on p. 64 of chapter 5.

Syntax

In the CREATE TABLE statement, you identify both the external file specification (location and file name) and the definition of the contained records as typed Firebird columns:

```
CREATE TABLE EXT_TBL EXTERNAL FILE filespec
(columndef [, columndef, ...],
, linedelimiter1 CHAR(1)[, linedelimiter2] CHAR(1));
• filespec is the fully qualified path and file specification for the external data file
```

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

- *columndef* is an ordinary Firebird column definition. Non-character data types can be specified, provided every string extracted from the column's location in the external record is capable of being cast implicitly to that type
- *linedelimiter* is a final column or pair of columns defined to read the characters used by the file system to mark the end of a line of text.
 - on Linux/UNIX, this is the single character ASCII 10, the linefeed character
 - on Windows, it is the ordered pair ASCII 13 (carriage return) followed by ASCII 10
 - other operating systems may use other variations or other characters

★ When INSERTing to an external file on Windows, the UDF Character() from the FreeUDFLib library can be used to pass the carriage return and linefeed characters to the line delimiter fields in the SQL statement.

On Linux/UNIX, the linefeed character (and others) can be copy/pasted interactively from an existing text file into a table of ascii characters and sub-selected into the external table in the SQL INSERT statement.

Uses for the EXTERNAL FILE option

Use the EXTERNAL FILE option to:

- Import data from a flat external file in a known fixed-length format into a new or existing Firebird table. This allows you to populate a Firebird table with data from an external source. Many applications allow you to create an external file with fixed-length records.
- SELECT from the external file as if it were a standard Firebird table.
- Export data from an existing Firebird table to an external file. You can format the data from the Firebird table into a fixed-length file that another application can use.

Restrictions

The following restrictions apply to using the EXTERNAL FILE option:

- The external file *must exist* before you try to read from or write to it from inside the database.
- Each record in the external file must be of fixed length, consisting of fixed-length fields.
- You cannot read from or write to BLOB or array data in an external file.
- When you create the table that will be used to import the external data, you must define a column to contain the end-of-line (EOL) or new-line character. The size of this column must be exactly large

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

enough to contain a particular system's EOL symbol (usually one or two bytes). For Macintosh and most versions of UNIX, it is 1 byte. For Windows it is 2 bytes.

- For some tips about catering for these characters, see the notes in the previous section, *Syntax*.
- You can only INSERT into and SELECT from the rows of an external table. UPDATE and DELETE are not supported and will return an error if attempted.
- Inserting into and selecting from an external table are not under standard transaction control because the external file is outside the database. Therefore, changes are immediate and permanent—you cannot roll them back.

❖ If you want your table to be under transaction control, create another internal Firebird table, and insert the data from the external table into the internal one.

- While it is possible to read in numeric data directly from an external table, it is often easier and more precise to read it in as character, and convert using the CAST() function.
- Data to be treated as VARCHAR in Firebird must be stored in an external file in the following format:
`<2-byte unsigned short><string of character bytes>`

where the 2-byte unsigned short indicates the number of bytes in the actual string, and the string immediately follows.

❖ Because it is not readily portable, using VARCHAR data in an external file is not recommended.

- If you use DROP DATABASE to delete the database, you must also remove the external file—it will not be automatically deleted as a result of DROP DATABASE.

Importing external files to Firebird tables

To import an external file into a Firebird table:

- 1 Create a Firebird table that allows you to view the external data. Declare all columns as CHAR. The text file containing the data must be on the server. In the following example, the external file exists on a UNIX system, so the EOL character is 1 byte.

```
CREATE TABLE EXT_TBL EXTERNAL FILE 'file.txt'  
(FNAME CHAR(10),  
LNAME CHAR(20),
```

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

```
HDATE CHAR(8),  
NEWLINE CHAR(1));  
COMMIT;
```

- 2 Create another Firebird table that will eventually be your working table. If you expect to export data from the internal table back to an external file at a later time, be sure to create a column to hold the linefeed. Otherwise, you do not need to leave room for the newline character. In the following example, a column for the newline is provided:

```
CREATE TABLE PEOPLE  
(FIRST_NAME CHAR(10),  
LAST_NAME CHAR(20),  
HIRE_DATE CHAR(10),  
NEW_LINE CHAR(1));  
COMMIT;
```

- 3 Create and populate the external file. You can create the file with a text editor, or you can create an appropriate file with an application like Paradox for Windows or dBASE for Windows. If you create the file yourself with a text editor, make each record the same length, pad the unused characters with blanks, and insert the EOL character(s) at the end of each record.

★ The number of characters in the EOL is platform-specific. You need to know how many characters are contained in your platform's EOL (typically one or two) in order to correctly format the columns of the tables and the corresponding records in the external file.

In the following example, the record length is 36 characters. “b” represents a blank space, and “#” represents the EOL:

```
123456789012345678901234567890123456  
fname.....lname.....hdate...#  
-----#  
Robert bbbbBrickman bbbbbb#6/12/92#  
Sam bbbbJones bbbbbb#12/13/93#
```

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

- 4 At this point, when you do a SELECT statement from table EXT_TBL, you will see the records from the external file:

```
SELECT FNAME, LNAME, HDATE FROM EXT_TBL;
```

FNAME	LNAME	HDATE
Robert	Brickman	12-JUN-1992
Sam	Jones	13-DEC-1993

- 5 Insert the data into the destination table.

```
INSERT INTO PEOPLE SELECT FNAME, LNAME, CAST(HDATE AS DATE),  
NEWLINE FROM EXT_TBL;
```

Now if you SELECT from PEOPLE, the data from your external table will be there.

```
SELECT FIRST_NAME, LAST_NAME, HIRE_DATE FROM PEOPLE;
```

FIRST_NAME	LAST_NAME	HIRE_DATE
Robert	Brickman	12-JUN-1992
Sam	Jones	13-DEC-1993

Firebird allows you to store a string of a recognized date literal format as a DATE type by converting from a CHAR(10) (in the above example) to DATE using the CAST() function.

Casting datatypes

If your application programming language does not support a particular datatype, you can let Firebird automatically convert the data to an equivalent datatype (an implicit type conversion), or you can use the CAST() function in search conditions to explicitly translate one datatype into another for comparison purposes. For more information about specifying datatypes, see chapter 13, [Firebird Data Types](#) (p. 231). For details about using the CAST() function, see [The CAST\(\) function](#) on page 254.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Exporting Firebird tables to an external file

If you add, update, or delete a record from an internal table, the changes will not be reflected in the external file. So in the previous example, if you delete the "Sam Jones" record from the PEOPLE table, and do a subsequent SELECT from EXT_TBL, you would still see the "Sam Jones" record.

This section explains how to export Firebird data to an external file. Using the example developed in the previous section, follow these steps:

- 1 Open the external file in a text editor and remove everything from the file. If you then do a SELECT on EXT_TBL, it should be empty.
- 2 Use an INSERT statement to copy the Firebird records from PEOPLE into the external file, `file.txt`.

```
INSERT INTO EXT_TBL SELECT FIRST_NAME, LAST_NAME, HIRE_DATE,  
      NEW_LINE FROM PEOPLE WHERE FIRST_NAME LIKE 'Rob%';
```

- 3 Now if you do a SELECT from the external table, EXT_TBL, only the records you inserted should be there. In this example, only a single record should be displayed:

```
SELECT FNAME, LNAME, HDATE FROM EXT_TBL;  
FNAME      LNAME          HDATE  
=====  ======  ======  
Robert    Brickman        12-JUN-1992
```

- ❖ Make sure that all records that you intend to export from the internal table to the external file have the correct EOL character(s) in the newline column.

Altering tables

A table can be altered by its creator, the SYSDBA user, and any users with operating system root privileges.

Use ALTER TABLE to modify the structure of an existing table. ALTER TABLE allows you to:

- Add a new column to a table.
- Drop a column from a table.

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

- Drop integrity constraints from a table or a column.
- Modify the column name, datatype, and position.

You can perform any number of the above operations with a single ALTER TABLE statement.

❖ Alterations to each table, or to its triggers, are reference-counted. Any one table can be altered at most 255 times before you must back up and restore the database.

★ In a Firebird database, switching a trigger on and off using

```
ALTER TRIGGER triggername ACTIVE | INACTIVE  
does not affect the reference count,
```

Preparing to use ALTER TABLE

Before modifying or dropping columns or attributes in a table, you need to do three things:

- 1 Make sure you have the proper database privileges.
- 2 Save the existing data.
- 3 Drop any dependency constraints on the column.

Saving existing data

Before modifying an existing column definition using ALTER TABLE, you must preserve existing data, or it will be lost. Preserving data in a column and modifying the definition for a column, is a five-step process:

- 1 Add a temporary column to the table whose definition mirrors the current column to be changed.
- 2 Copy the data from the column to be changed to the temporary column, e.g.

```
UPDATE aTable  
SET tempcol = oldcol;
```

- 3 Modify the temporary column.
- 4 Copy the data from the temporary column to the old column, e.g.

```
UPDATE aTable  
SET oldcol = tempcol;
```

- 5 Drop the temporary column.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

An example

Suppose the EMPLOYEE table contains a column, OFFICE_NO, defined as CHAR(3), and suppose that the size of the column needs to be increased by one. The following sequence describes a scenario and steps through a recommended procedure:

1 First, create a temporary column to hold the data from OFFICE_NO during the modification process:

```
ALTER TABLE EMPLOYEE ADD TEMP_NO CHAR(3);
```

2 Move existing data from OFFICE_NO to TEMP_NO to preserve it:

```
UPDATE EMPLOYEE  
SET TEMP_NO = OFFICE_NO;
```

3 Modify TEMP_NO, specifying the datatype and new size:

```
ALTER TABLE ALTER TEMP_NO TYPE CHAR(4);
```

4 Move the data from TEMP_NO to OFFICE_NO:

```
UPDATE EMPLOYEE  
SET OFFICE_NO = TEMP_NO;
```

5 Finally, drop the TEMP_NO column:

```
ALTER TABLE DROP TEMP_NO;
```

Dropping columns

Before attempting to drop or modify a column, you should be aware of the different ways that ALTER TABLE can fail:

- The person attempting to alter data does not have the required privileges.
- Current data in a table violates a PRIMARY KEY or UNIQUE constraint definition added to the table; there is duplicate data in columns that you are trying to define as PRIMARY KEY or UNIQUE.
- The column to be dropped is part of a UNIQUE, PRIMARY, or FOREIGN KEY constraint.
- The column is used in a CHECK constraint. When altering a column based on a domain, you can supply an additional CHECK constraint for the column. Changes to tables that contain CHECK constraints with subqueries can cause constraint violations.
- The column is used in another view, trigger, or in the value expression of a computed column.

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

- ❖ You must drop the constraint or computed column before dropping the table column. PRIMARY KEY and UNIQUE constraints cannot be dropped if they are referenced by FOREIGN KEY constraints. In this case, drop the FOREIGN KEY constraint before dropping the PRIMARY KEY or UNIQUE key it references. Finally, you can drop the column.
- ❖ When you alter or drop a column, all data stored in it are lost.

Using ALTER TABLE

ALTER TABLE allows you to make the following changes to an existing table:

- Modify column names, datatypes, and position
- Add new column definitions.
 - ★ To create a column using an existing name, you must drop or rename the existing column definition and commit the change before adding new ones.
- Add new table constraints. To create a constraint using an existing name, you must drop existing constraints with that name before adding a new one.
- Drop existing column definitions without adding new ones.
- Drop existing table constraints without adding new ones.

Syntax for adding a new column to a table

The syntax for adding a column with ALTER TABLE is:

```
ALTER TABLE table ADD <col_def>

<col_def> = col {<datatype> | [COMPUTED [BY] (<expr>) | domain}
    [DEFAULT {literal | NULL | USER}]
    [NOT NULL] [<col_constraint>]
    [COLLATE collation]

<col_constraint> = [CONSTRAINT constraint] <constraint_def>
    [<col_constraint>]
```

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

<constraint_def> =

PRIMARY KEY

| UNIQUE
| CHECK (<search_condition>)
| REFERENCES other_table [(other_col [, other_col ...])]
[ON DELETE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
[ON UPDATE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]

For the complete syntax of ALTER TABLE, see the *Firebird Reference Guide*.

Examples

The following statement adds a column, EMP_NO, to the EMPLOYEE table using the EMPNO domain:

```
ALTER TABLE EMPLOYEE ADD EMP_NO EMPNO NOT NULL;
```

Multiple columns can be added to a table in a single statement, separating column definitions with commas. The following statement adds two columns, EMP_NO, and FULL_NAME, to the EMPLOYEE table. FULL_NAME is a computed column, a column that derives its values from calculations based on two other columns already defined for the EMPLOYEE table:

```
ALTER TABLE EMPLOYEE
  ADD EMP_NO EMPNO NOT NULL,
  ADD FULL_NAME COMPUTED BY (LAST_NAME || ' ', ' ' || FIRST_NAME);
```

Including integrity constraints

Integrity constraints can be included for columns that you add to the table. For example, the next statement adds two columns, CAPITAL and LARGEST_CITY, to the COUNTRY table, and defines a UNIQUE constraint on CAPITAL:

```
ALTER TABLE COUNTRY
  ADD CAPITAL VARCHAR(25) UNIQUE,
  ADD LARGEST_CITY VARCHAR(25) NOT NULL;
```

Adding new table constraints

You can use ALTER TABLE to add a new table-level constraint. The syntax is:

```
ALTER TABLE name ADD [CONSTRAINT constraint] <tconstraint_opt>;
```

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

where *tconstraint_opt* is a PRIMARY KEY, FOREIGN KEY, UNIQUE, or CHECK constraint. For example:

```
ALTER TABLE EMPLOYEE  
ADD CONSTRAINT DEPT_NO UNIQUE(PHONE_EXT);
```

Dropping an existing column from a table

You can use ALTER TABLE to drop (remove) a column definition and its data from a table.

- A column can be dropped only by the owner of the table.
- If another user is accessing a table when you attempt to drop a column, the other user's transaction will continue to have access to the table until that transaction completes. Firebird postpones the drop until the table is no longer in use.

Syntax

The syntax for dropping a column with ALTER TABLE is:

```
ALTER TABLE name DROP colname [, colname ...];
```

For example, the following statement drops the EMP_NO column from the EMPLOYEE table:

```
ALTER TABLE EMPLOYEE DROP EMP_NO;
```

Multiple columns can be dropped with a single ALTER TABLE statement:

```
ALTER TABLE EMPLOYEE  
DROP EMP_NO,  
DROP FULL_NAME;
```

- ❖ A column that is part of a UNIQUE, PRIMARY KEY, or FOREIGN KEY constraint cannot be dropped. In the previous example, EMP_NO is the PRIMARY KEY for the EMPLOYEE table, so you cannot drop this column unless you first drop the PRIMARY KEY constraint.

Dropping existing constraints from a column

Constraints must be dropped from a column in the correct sequence.

FOREIGN KEY and PRIMARY KEY constraints

See the following CREATE TABLE example. Because there is a foreign key in the PROJECT table that references the primary key (EMP_NO) of the EMPLOYEE table, the foreign key reference must be dropped from this table before the PRIMARY KEY constraint can be dropped from the EMPLOYEE table.

**Symbols** [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

```
CREATE TABLE PROJECT
  (PROJ_ID PROJNO NOT NULL,
   PROJ_NAME VARCHAR(20) NOT NULL UNIQUE,
   PROJ_DESC BLOB(800,1),
   TEAM_LEADER EMPNO,
   PRODUCT PRODTYPE,
   PRIMARY KEY (PROJ_ID),
   CONSTRAINT TEAM_CONSTRT FOREIGN KEY (TEAM_LEADER) REFERENCES
   EMPLOYEE (EMP_NO));
```

The proper sequence is:

```
ALTER TABLE PROJECT
  DROP CONSTRAINT TEAM_CONSTRT;

ALTER TABLE EMPLOYEE
  DROP CONSTRAINT EMP_NO_CONSTRT;

ALTER TABLE EMPLOYEE
  DROP EMP_NO;
```

CHECK constraints

If a column is referred to by another column's CHECK constraint, it cannot be dropped. To drop the column, first drop the CHECK constraint, then drop the column.

★ Constraint names can be found in the system table, RDB\$RELATION_CONSTRAINTS.

Modifying columns in a table

Syntax

```
ALTER TABLE table ALTER [COLUMN]simple_column_name alter_rel_field
alter_rel_field = new_col_name | new_col_type | new_col_pos
new_col_name = TO simple_column_name
new_col_type = TYPE datatype_or_domain
new_col_pos = POSITION integer
```

**Symbols** [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

For the complete syntax, see *Firebird Reference Guide*—[ALTER TABLE](#) (ch. 2 p. 54).

Examples

The following statement moves a column, EMP_NO, from the third position to the second position in the EMPLOYEE table:

```
ALTER TABLE EMPLOYEE ALTER EMP_NO POSITION 2;
```

The name of the EMP_NO column is changed to EMP_NUM in the following example:

```
ALTER TABLE EMPLOYEE ALTER EMP_NO TO EMP_NUM;
```

The data type of the EMP_NUM column is changed to CHAR(20):

```
ALTER TABLE EMPLOYEE ALTER EMP_NUM TYPE CHAR(20);
```

Conversions from non-character to character data will be allowed with the following restrictions:

- Blob and array types are not convertible.
 - Field types (character or numeric) cannot be shortened.
 - The new field definition must be able to hold the existing data (for example, the new field has too few CHARs or the datatype conversion is not supported) or an error is returned.
 - Conversions from character data to non-character data are not allowed.
- ❖ Any changes to the field definitions may require the indexes to be rebuilt.

Constraints on altering data types

Firebird does not let you alter the data type of a column or domain in a way that might result in data loss.

Converting a numeric data type to a character type requires a minimum length for the character type as tabulated below:

TABLE 17-6 Minimum character lengths for numeric conversions

data type	Minimum length for converted character type
Decimal	20
Double	22
Float	13



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

TABLE 17–6 Minimum character lengths for numeric conversions

data type	Minimum length for converted character type
Integer	11
Numeric	22
Smallint	6

The following table lists valid conversions:

TABLE 17–7 Valid datatype conversions using ALTER COLUMN and ALTER DOMAIN

Convert from:	Convert to:											
	Blob	Char	Date	Decimal	Double	Float	Integer	Numeric	Timestamp	Time	Smallint	Varchar
Blob												
Char		X										X
Date	X	X								X		
Decimal	X			X				X				X
Double	X				X	X						X
Float	X				X	X						X
Integer	X		X	X			X	X				X
Numeric	X							X				X



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

TABLE 17-7 Valid datatype conversions using ALTER COLUMN and ALTER DOMAIN

Convert to:		Blob	Char	Date	Decimal	Double	Float	Integer	Numeric	Time-stamp	Time	Smallint	Varchar
Convert from:													
Timestamp	X		X							X	X		
Time	X									X	X		
Smallint	X			X	X	X	X	X	X			X	X
Varchar	X												X

Summary of ALTER TABLE arguments

When you use ALTER TABLE to add column definitions and constraints, you can specify all of the same arguments that you use in CREATE TABLE; all column definitions, constraints, and data type arguments are the same, with the exception of the *operation* argument. The following operations are available for ALTER TABLE.

- Add a new column definition with ADD *col_def*
- Add a new table constraint with ADD *table_constraint*
- Drop an existing column with DROP *colname*
- Drop an existing constraint with DROP CONSTRAINT *constraint*
- Modify column names, datatypes, and positions with ALTER [COLUMN] *colname*

Removing tables

Use DROP TABLE to remove an entire table permanently from the database.

- ❖ If you merely want to drop (remove) columns from a table, use ALTER TABLE.

**Symbols** [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

A table can be dropped by its creator, the SYSDBA user, or any user with operating system root privileges. An attempt by any other user to drop a table will return an error.

Dropping a table

Use DROP TABLE to remove a table's data, metadata, and indexes from a database. It also drops any triggers that are based on the table.

A table cannot be dropped if

- it is referenced in a computed column, a view, integrity constraint, or stored procedure. The dependencies should be removed before any attempt is made to drop the table
 - it is being used by an active transaction. The drop is postponed until the table is no longer in use
 - it has a UNIQUE or PRIMARY KEY defined for it, and the PRIMARY KEY is referenced by a FOREIGN KEY in another table. First drop the FOREIGN KEY constraints in the other table, then drop the table.
 - it is in use when the drop is attempted. The drop is postponed until the table is no longer in use
 - it is referenced in another table's CHECK constraint.
- ❖ DROP TABLE does not delete external tables; it removes the table definition from the database. You must explicitly delete the external file.

DROP TABLE syntax

```
DROP TABLE name;
```

The following statement drops the table, COUNTRY:

```
DROP TABLE COUNTRY;
```

The RECREATE TABLE statement

Sometimes, you may want to drop a table and create it again "from scratch". For these occasions, Firebird has RECREATE TABLE.

RECREATE TABLE does the following:

- drops the existing table and commits the change
- creates the new table as specified in the clauses and sub-clauses



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

Syntax

The syntax is identical to that of CREATE TABLE. Simply substitute the CREATE keyword with RECREATE and proceed.

For CREATE TABLE syntax and rules—

- see [Creating tables](#) on page 315 in this chapter
- refer to the *Firebird Reference Guide*— [SQL Statement and Function Reference](#) (ch. 2 p. 43) for the detailed syntax specification for [RECREATE TABLE](#)

Restrictions

The same restrictions apply to RECREATE TABLE as to DROP TABLE. For a summary, see [Removing tables](#) on page 349 of this chapter.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

CHAPTER 18

Indexes

Just as you search an index in a book as a quick way to locate the page numbers pointing to a topic that you want to read, a table index serves as a logical pointer to the physical locations (addresses) of a specified value among the pages stored in the database for a particular table. Indexes—sometimes pluralized as *indices*—accelerate the retrieval of rows meeting certain search conditions and the sorting of rows into ordered sets. They also provide the mechanism for enforcement of rules such as uniqueness and referential integrity.

Index basics

An index stores each value of the indexed column or columns along with pointers to all of the disk blocks that contain rows with that column value.

When a query is to be executed, a set of procedures in the Firebird engine—known as the optimizer—kicks in and begins by checking for the existence of indexes for the tables involved. It uses a number of algorithms to determine whether it is more efficient to scan the entire table or to use a suitable index to process the query.

If the engine decides to use an index, it searches the pages of the index to find the key values required and follows the pointers to locate the indicated rows in the table data pages. Data retrieval is fast because the values in the index are ordered and the index is relatively small. This allows the system to locate wanted values directly, by pointer, and to avoid visiting unwanted rows at all. Using an index typically requires fewer page fetches than “walking through” every row in the table.

An index can be defined on a single column of a table or it can span multiple columns. Multi-column indexes—also known as *composite* or *complex* indexes—can be utilized for single-column searches, as long as the column that is being searched is the *first* in the index.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

When to index

The length of time it takes to search a whole table is directly proportional to the number of rows in the table. An index on a column can mean the difference between an immediate response to a query and a long wait. So— why not index every column?

The main drawbacks are that indexes consume additional disk space, and inserting, deleting, and updating rows takes longer on indexed columns than on non-indexed columns. The index must be updated each time a data item in the indexed column changes and each time a row is added to or deleted from the table.

Nevertheless, the boost in performance for data retrieval queries usually outweighs the overhead of index maintenance. You should create an index on a column when:

- Search conditions frequently reference the column.
- Join conditions frequently reference the column.
- ORDER BY clauses frequently use the column to sort data.

You do *not* need to create an index for:

- Columns that are seldom referenced in search conditions.
- Frequently updated non-key columns.
- Columns that have a small number of possible values.

Use indexes to improve speed of data access when queries require

- ordered sets—When sets must be ordered on multiple columns, composite indexes which reflect the output order specified in ORDER BY clauses will improve retrieval speed exponentially
 - aggregation—Single-column or suitably ordered complex indexes will improve the speed with which aggregations are formed with complex GROUP BY clauses
 - searching of columns specified in WHERE clauses
 - when evaluating strings for exact matches or using STARTING WITH and CONTAINING predicates
- ★ Indexes are not used for searches predicated with LIKE, nor for *set-based searches* using the IN(..) operator.
- when performing direct comparison operations or BETWEEN evaluations
 - searched updates or deletes, i.e. when these operations use a WHERE clause



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Creation of indexes

Indexes are created either deliberately by a database developer, using the CREATE INDEX statement, or automatically by the system, as a side effect of the definition of keys in [RE]CREATE TABLE and ALTER TABLE statements. Firebird allows up to 64 user-created indexes on a given table. To create indexes, a user must be authorized to connect to the database.

Inspecting indexes

To inspect all indexes defined in the current database, use the **isql** command SHOW INDEX. To see all indexes defined for a specific table, use the command, SHOW INDEX *tablename*. To view information about a specific index, use SHOW INDEX *indexname*.



For more information, see [SHOW INDEX \(SHOW INDICES\)](#) on p. 172 of chapter 10 .

Automatic indexing

Firebird automatically generates ascending indexes on a column or set of columns when PRIMARY KEY, FOREIGN KEY, and UNIQUE constraints are defined for a table. These indexes preserve referential integrity.

Indexes on PRIMARY KEY columns

Firebird imposes a UNIQUE constraint on the column or group of columns forming the primary key of a table. Columns participating in a primary key cannot contain NULL. Hence, by design, the index which is automatically created for a primary key will be unique and non-nullable and will enforce these rules when new rows are inserted or if any column in the primary key is modified.

Indexes on FOREIGN KEY columns

The index which Firebird creates automatically on the column or columns participating a foreign key is non-unique and allows NULLs on any participating columns that do not carry the NOT NULL attribute. Nullable foreign keys columns raise the possibility for orphan rows to exist in the table, regardless of referential integrity—see [Orphan rows](#) on p. 330 of chapter 17.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Indexes on UNIQUE-constrained columns

Firebird automatically creates a unique index for a column or group of columns to which a UNIQUE constraint is applied. Like a primary key index, this index cannot contain NULL in any of its elements. A UNIQUE constraint can be referenced as the "parent" key of a FOREIGN KEY constraint in another table.

- ★ A unique index and a UNIQUE column constraint are not the same thing. Placing a *unique index* on a column does not make it a *unique key* for referential purposes. For information about defining a UNIQUE (key) constraint, see [UNIQUE keys](#) on p. 326 of chapter 17.

Duplicating system indexes

Although Firebird allows you to create indexes whose specifications match those it creates automatically for key constraints, it is **strongly** recommended that you AVOID duplicating indexes. In some of its algorithms, the Firebird optimizer will treat duplicated indexes as ambiguous and ignore both of them.

Using CREATE INDEX

The CREATE INDEX statement creates an index on one or more columns of a table. A single-column index searches only one column in response to a query, whereas a multi-column index searches one or more columns. Parameters specify:

- The sort order for the index.
- Whether duplicate values are allowed in the indexed column.

Syntax

```
CREATE [UNIQUE] [ASC[ENDING] | DESC[ENDING]]  
INDEX index ON table (col [, col ...]);
```

Preventing duplicate entries

No two rows can be alike when a UNIQUE index is specified for a column or set of columns. The system checks for duplicate values when the index is created, and each time a row is inserted or updated. Unique indexes make sense only when uniqueness is an intrinsic characteristic of the data items to be stored in its column or columns.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

For example, you would not define a unique index on a LAST_NAME column because the probability of duplication is high. Conversely, a unique index is a good idea on a column containing a social security number.

Existing duplicates

Firebird does not allow you to create a unique index on a column that already contains duplicate values. Before defining a UNIQUE index, use a SELECT statement to ensure there are no duplicate keys in the table. For example:

```
SELECT PRODUCT, PROJ_NAME FROM PROJECT
GROUP BY PRODUCT, PROJ_NAME
HAVING COUNT(*) > 1;
```

Defining a unique index

To define an index that disallows duplicate entries, include the UNIQUE keyword in CREATE INDEX. The following statement creates a unique ascending index (PRODTYPEX) on the PRODUCT and PROJ_NAME columns of the PROJECT table:

```
CREATE UNIQUE INDEX PRODTYPEX ON PROJECT (PRODUCT, PROJ_NAME);
```

- ★ A unique index and a UNIQUE column constraint are not the same thing. Placing a *unique index* on a column does not make it a *unique key* for referential purposes. For information about defining a UNIQUE (key) constraint, see [UNIQUE keys](#) on p. 326 of chapter 17.

Specifying index sort order

The keywords ASCENDING or DESCENDING are used to specify the sort order of an index. By default, Firebird creates indexes in ascending order. To make a descending index on a column or group of columns, use the DESC[ENDING] keyword to define the index. The following statement creates a descending index (DESC_X) on the CHANGE_DATE column of the SALARY_HISTORY table:

```
CREATE DESCENDING INDEX DESC_X ON SALARY_HISTORY (CHANGE_DATE);
```

- ★ To retrieve indexed data from this table in descending order, use ORDER BY CHANGE_DATE DESCENDING in the SELECT statement. The optimizer will use the descending index if it exists.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

If you intend to use both ascending and descending sort orders on a particular column, define both an ascending and a descending index for the same column. The following example illustrates this:

```
CREATE ASCENDING INDEX ASCEND_X ON SALARY_HISTORY (CHANGE_DATE);  
CREATE DESCENDING INDEX DESC_X ON SALARY_HISTORY (CHANGE_DATE);
```

When to use a multi-column index

If your applications frequently need to search, order or group on the same group of multiple columns in a particular table, it will be of benefit to create a multi-column index (also known as a *composite* or *complex* index). Queries do not need to be constructed with the exact column list that is defined in the index. Firebird will use a subset of the components of a multi-column index to optimize a query if

- the subset of columns used in the ORDER BY clause begins with the first column in the multi-column index. Firebird cannot use a single element of composite index to optimize a search unless all of the preceding elements of the index are also used. For example, if the index column list is A1, A2, and A3, a query using A1 and A2 would be optimized using the index, but a query using A2 and A3 would not.
- the *left-to-right order* in which the query accesses the columns in an ORDER BY clause matches the left-to-right order of the column list defined in the index. (The query would not be optimized if its column list were A2, A1.)

OR predicates in queries

If you expect to issue frequent queries against a table where the queries use the OR operator, it is better to create a single-column index for *each* condition. Since multi-column indices are sorted hierarchically, a query that is looking for any one of two or more conditions would, of course, have to search the whole table, losing the advantage of an index.

Examples

The first example creates a multi-column index, NAMEX, on the EMPLOYEE table:

```
CREATE INDEX NAMEX ON EMPLOYEE (LAST_NAME, FIRST_NAME);
```

The following query will be optimized against the index because the ORDER BY clause references all of the indexed columns in the correct order:

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

```
SELECT LAST_NAME, SALARY FROM EMPLOYEE  
WHERE SALARY > 40000  
ORDER BY LAST_NAME, FIRST_NAME;
```

The next query will also process the following query with an index search (using LAST_NAME from NAMEX) because although the ORDER BY clause only references one of the indexed columns (LAST_NAME), it does so in the correct order.

```
SELECT LAST_NAME, SALARY FROM EMPLOYEE  
WHERE SALARY > 40000  
ORDER BY LAST_NAME;
```

Conversely, the following query will *not* be optimized against the index because the ORDER BY clause uses FIRST_NAME, which is not the first indexed column in the NAMEX column list.

```
SELECT LASTNAME, SALARY FROM EMP  
WHERE SALARY > 40000  
ORDER BY FIRST_NAME;
```

The same rules that apply to the ORDER BY clause also apply to queries containing a WHERE clause. The next example creates a multi-column index for the PROJECT table:

```
CREATE UNIQUE INDEX PRODTYPEX ON PROJECT (PRODUCT, PROJ_NAME);
```

The following query will be optimized against the PRODTYPEX index because the WHERE clause references the first indexed column (PRODUCT) of the index:

```
SELECT * FROM PROJECT  
WHERE PRODUCT = 'software';
```

Conversely, the next query will not be optimized against the index because PROJ_NAME is not the first indexed column in the column list of the PRODTYPEX index:

```
SELECT * FROM PROJECT  
WHERE PROJ_NAME STARTING WITH 'Firebird 1';
```



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Improving index performance

Because indexes play such an important part in the performance of a database, it is important to recognize that such dynamic structures need to be subjected to housekeeping from time to time. Also, under some conditions, it may be of benefit to disable indexes for certain types of operation.

Housekeeping

Indexes are binary structures which tend to become unbalanced after many changes to the database, especially if general database housekeeping is neglected. Indexes can be rebalanced and tuned in a number of ways to restore performance to optimal levels.

- Rebuilding an index will restore the balance of its tree structure by removing entries made obsolete by deletions and redistributing branches created by successive insertions.
 - The tool for switching the state of an index between active and inactive is the ALTER INDEX statement.
 - An index may also be completely rebuilt by using DROP INDEX to remove it and CREATE INDEX to build it afresh. *This is necessary when you need to change the structure of an index.*
- Recomputing index selectivity updates the selectivity value stored in the system tables—a reciprocal which indicates the nearness of the index to uniqueness. The smaller the value stored, the closer to uniqueness. The optimizer reads this just once when choosing a plan. Selectivity is largely irrelevant in optimizer's construction of a plan; but an accurate calculation can assist you in determining the usefulness or otherwise of the index. For example, any indexes whose selectivity reciprocal tends to rise dramatically over time should be dropped because of their ecological effects on performance.
 - The tool for recomputing index selectivity is the SET STATISTICS statement.
- A backup and restore of the database using **gbak** will both rebuild all indexes and cause their selectivity to be freshly computed.

Using ALTER INDEX

The ALTER INDEX statement deactivates and reactivates an index. Deactivating and reactivating an index is useful when changes in the distribution of indexed data cause the index to become unbalanced.

To rebuild the index, first use ALTER INDEX INACTIVE to deactivate the index, then ALTER INDEX ACTIVE to reactivate it again. This method recreates and balances the index.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Deactivating for batch inserts

Before inserting a large batch of rows, deactivate a table's indexes during the insert, then reactivate the index to rebuild it. This avoids the overhead of Firebird maintaining the index upon each individual insertion.

Syntax

The syntax for ALTER INDEX is:

```
ALTER INDEX name {ACTIVE | INACTIVE};
```

The following statements deactivate and reactivate an index to rebuild it:

```
ALTER INDEX BUDGETX INACTIVE;
```

```
ALTER INDEX BUDGETX ACTIVE;
```

Restrictions

- An index can be altered by the creator of the index, the SYSDBA user, or (on Linux/UNIX) a user with operating system root privileges.
 - If it is *in use* in an active database, an index cannot be altered. All requests using an index must be released to make it available.
 - An index is *in use* if it is currently being used by a dynamic or compiled request to process a query.
 - An index that was defined automatically to support a UNIQUE, PRIMARY KEY, or FOREIGN KEY constraint cannot be altered. ALTER TABLE must be used to modify or delete the constraints.
- For more information about ALTER TABLE, see [Preparing to use ALTER TABLE](#) on p. 341 of chapter 17. For detailed syntax, see the *Firebird Reference Guide*—[ALTER INDEX](#) (ch. 2 p. 51).
- ALTER INDEX cannot be used to alter the structure of index columns. Use DROP INDEX to delete the index and then redefine it with CREATE INDEX.

Using SET STATISTICS

In some tables, the number of duplicate values in indexed columns can increase or decrease radically as a result of the relative “popularity” of a particular value in the index when compared with other candidate values. Periodically recomputing *index selectivity* can improve performance. SET STATISTICS recomputes the selectivity of an index.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Index selectivity

Index selectivity is a calculation that the Firebird optimizer performs when a table is accessed. It is a factor of the number of distinct occurrences of a particular value in the index when divided by the number of rows in the table. As a generic observation, the smaller the number of distinct values actually in use in the index, the lower the selectivity of the index. Indexes with higher selectivity perform better than those with low selectivity. The selectivity factor is cached in memory, where the optimizer can access it to calculate the optimal retrieval plan for a given query.

Syntax

The syntax for SET STATISTICS is:

```
SET STATISTICS INDEX name;
```

The following statement recomputes the selectivity for an index:

```
SET STATISTICS INDEX MINSALX;
```

Restrictions

- SET STATISTICS is authorized to the creator of the index, the SYSDBA user, or (on Linux/UNIX) a user with operating system root privileges.
- SET STATISTICS on its own will not cure current problems resulting from previous index maintenance that depended on an obsolete selectivity factor, since it does not rebuild an index. To rebuild an index, use ALTER INDEX..

Using DROP INDEX

DROP INDEX removes a *user-defined* index from the database.

- ❖ System-defined indexes, such as those created on columns defined with UNIQUE, PRIMARY KEY, and FOREIGN KEY constraints, cannot be dropped.

Use DROP INDEX when you need to alter the structure of an index: add, remove or change columns in the index, or change the column or sort order. First use the DROP INDEX statement to delete the index, then use the CREATE INDEX statement to recreate the index (using the same name) with the desired characteristics.



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

Syntax

```
DROP INDEX name;
```

The following statement deletes an index:

```
DROP INDEX MINSALX;
```

Restrictions

- An index can be dropped by the creator of the index, the SYSDBA user, or (on Linux/UNIX) a user with operating system root privileges.
 - An index in use cannot be dropped until it is no longer in use. Attempting to alter or drop an index while transactions are being processed will have different results, according to the lock setting of the active transaction—
 - In a WAIT transaction, the ALTER INDEX or DROP INDEX operation waits until the index is not in use.
 - In a NOWAIT transaction, Firebird returns an error.
 - An index that was system-defined automatically to support a UNIQUE, PRIMARY KEY, or FOREIGN KEY constraint cannot be dropped. ALTER TABLE must be used to modify or delete the constraints.
- For more information about ALTER TABLE, see [Preparing to use ALTER TABLE](#) on p. 341 of chapter 17. For detailed syntax, see the *Firebird Reference Guide*—[ALTER TABLE](#) (ch. 2 p. 54).

Using gbak for index cleanup

When a database is backed up with the **gbak** utility, **gbak** stores only the definition the indexes, not their data. When the database is restored, **gbak** rebuilds all of the indexes afresh.

For details of how to backup and restore a database, see chapter 21, [Database Backup and Restore](#) (p. 390).



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

CHAPTER 19

Views

In SQL-89 and SQL-92 terms views are a standard table type which it also refers to as *viewed* or *virtual tables*. A views is characterized as *virtual* because, instead of storing its structure and allocating pages in which to store its data, the Firebird engine stores a metadata description which includes

- a unique name (identifier),
- a list of column names
- a statement by which to derive data for the view from persistent base tables

What is a view?

At its simplest, a view is a special kind of SELECT query specification which returns a data set that behaves in many ways like a persistent table. When a view is created, the Firebird engine physically stores its specifications and includes its output column definitions in the system tables, as if it were a persistent table.

A view's columns can be derived from one or more tables, from other views, from aggregations and sub-selects and from calculations. Once a view is defined, you can display and operate on it as if it were an ordinary table—selecting rows from it, deriving further aggregations and calculated columns into its output, applying selection criteria to the output columns, and so on. In many cases, views are updatable, or can be made so—insert, update and delete operations can be applied to a view and cause modifications to the table from which its columns originate.

 For more information about query specifications, refer to the chapter [Firebird SQL & Queries](#).

For more information about modifying table data through views, see [Naturally updatable views](#) and [Making read-only views updatable](#).

A view is a virtual table—a “window” through which actual data are exposed. Creating one does not generate a copy of the data for storage elsewhere: the data pertaining to a view remains stored in the underlying tables.

**Symbols** [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

When the tables' data change, the data content of the view changes with it. When data are changed through a DML operation on a view, underlying tables' data change accordingly.

Furthermore, a view is a database object , meaning that it requires specific user privileges in order to be accessed. Through granting privileges to a view, it is possible to provide users with very finely-grained access to certain columns and rows from tables, whilst denying them access to other, more sensitive data stored in the underlying tables.



For more information about granting user privileges, see [Database-level security](#) on p. 429 of chapter 22.

Why views can be useful

The data requirements of an individual user or user group are often quite consistent. Views provide the means to create custom versions of the underlying tables to target clusters of data that are pertinent to specific users and their tasks. To summarize the benefits of views:

- **Simplified access to the data, object re-use**

Views enable you to encapsulate a subset of data from one or more tables to use as a foundation for future queries wherever the alternative is to repeat the same set of SQL statements to retrieve the same subset of data.

- **Customized access to the data**

Views provide a way to tailor the database output so that it is task-oriented, suits the specific skills and requirements of users and avoids moving extraneous data across networks.

- **Data independence**

Views can be a device to shield user applications from the effects of changes to database structure. For example, if the database administrator decides to split one table into two, a view can be created that is a join of the two new tables, which applications can continue to query as if it were still a single, persistent table.

- **Data security**

Views can restrict access to sensitive or irrelevant portions of tables. For example, a user might be able to look up job information through a view over an Employee table, without seeing associated salary information.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Ways to derive views

A view can be created from:

- **A vertical subset of columns from a single table** For example, the table, JOB, in the employee.gdb database has 8 columns: JOB_CODE, JOB_GRADE, JOB_COUNTRY, JOB_TITLE, MIN_SALARY, MAX_SALARY, JOB_REQUIREMENT, and LANGUAGE_REQ. The following view displays a list of salary ranges (subset of columns) for all jobs (all rows) in the JOB table:

```
CREATE VIEW JOB_SALARY_RANGES AS  
    SELECT JOB_CODE, MIN_SALARY, MAX_SALARY FROM JOB;
```

- **A horizontal subset of rows from a single table** The next view displays all of the columns in the JOB table, but only the subset of rows where the MAX_SALARY is less than \$15,000:

```
CREATE VIEW LOW_PAY AS  
    SELECT * FROM JOB  
    WHERE MAX_SALARY < 15000;
```

- **A combined vertical and horizontal subset of columns and rows from a single table** The next view displays only the JOB_CODE and JOB_TITLE columns and only those jobs where MAX_SALARY is less than \$15,000:

```
CREATE VIEW ENTRY_LEVEL_JOBS AS  
    SELECT JOB_CODE, JOB_TITLE FROM JOB  
    WHERE MAX_SALARY < 15000;
```

- **A subset of rows and columns from multiple tables (joins)** The next example shows a view created from both the JOB and EMPLOYEE tables. The EMPLOYEE table contains 11 columns: EMP_NO, FIRST_NAME, LAST_NAME, PHONE_EXT, HIRE_DATE, DEPT_NO, JOB_CODE, JOB_GRADE, JOB_COUNTRY, SALARY, FULL_NAME. It displays two columns from the JOB table, and two columns from the EMPLOYEE table, and returns only the rows where SALARY is less than \$15,000:

```
CREATE VIEW ENTRY_LEVEL_WORKERS AS  
    SELECT JOB_CODE, JOB_TITLE, FIRST_NAME, LAST_NAME  
    FROM JOB, EMPLOYEE  
    WHERE JOB.JOB_CODE = EMPLOYEE.JOB_CODE AND SALARY < 15000;
```



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Creating views

The CREATE VIEW statement creates a virtual table with columns derived from one or more underlying tables. All styles of joined output sets supported in queries are supported, as are union sets. You can perform select, project, join, and union operations on views just as if they were tables.

- ❖ It is not possible to define a view that is derived from the output set of a stored procedure.

View privileges

Owner

The creator of the view must have the following privileges:

- To create a read-only view, the creator needs SELECT privileges for any underlying tables.
- To create an updatable view, the creator needs ALL privileges to the underlying tables.

The user who creates a view is its owner and has all privileges for it, including the ability to GRANT privileges to other users, triggers, and stored procedures.

User

The creator of the view must grant the appropriate privileges to users, stored procedures and other views needing to access the view. A user can be granted privileges to a view without having access to its base tables. In the case of updatable views, INSERT, UPDATE and DELETE privileges must be assigned if users of the view need to perform DML on underlying tables. Conversely, do not grant these privileges to users if your intention is to provide a read-only view.

- ❖ For more information on SQL privileges, see [Database-level security](#) on p. 429 of chapter 22.

Syntax

The syntax for CREATE VIEW is:

```
CREATE VIEW name [(view_col [, view_col ...])]  
AS <select> [WITH CHECK OPTION];
```

Specifying view column names

view_col names one or more columns for the view.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z Considerations

- Because the view's column name list specification must correspond in order and number to the columns listed in the SELECT statement, it is necessary specify either
 - a complete list of view column names. A *view_col* definition can contain one or more columns calculated from expressions. A complete column list is mandatory if the view includes columns calculated from expressions;
 - or
 - no list of column names at all. This is possible if the output set contains only data columns and, in the case of joins, properly aliasing of any output columns with duplicate names.
- Column names must be unique among all column names in the view. If column names are not specified, the view takes the column names from the underlying table by default.

Using the SELECT statement

The SELECT statement specifies the selection criteria for the rows to be included in the view.

- ❖ When creating views, the SELECT statement cannot include an ORDER BY clause.

SELECT does the following:

- Lists the columns to be included from the base table. When SELECT * is used rather than a column list, the view contains all columns from the base table, and displays them in the order in which they appear in the base table. The following example creates a view, MY_VIEW, that contains all of the columns in the EMPLOYEE table:

```
CREATE VIEW MY_VIEW AS
    SELECT * FROM EMPLOYEE;
```

- Identifies the source tables in the FROM clause. In the MY_VIEW example, EMPLOYEE is the source table.
- Specifies, if needed, row selection conditions in a WHERE clause. In the next example, only the employees that work in the USA are included in the view:

```
CREATE VIEW USA_EMPLOYEES AS
    SELECT * FROM EMPLOYEE
    WHERE JOB_COUNTRY = 'USA';
```

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

- If WITH CHECK OPTION is specified, any INSERT or UPDATE operation on an updatable view will be blocked if the operation would result in a violation of the search condition specified in the WHERE clause.

For more information about using this option, see [Using WITH CHECK OPTION](#) on page 372.

For an explanation of updatable views, see [Read-only and updatable views](#) on page 368.

Using expressions to define columns

Any expression that performs a comparison or computation and returns a single value can be involved in the specification for a view column.

Examples of expressions are concatenating character strings, performing computations on numeric data, doing comparisons using comparison operators (<, >, <=, and so on) or Boolean operators (AND, OR, NOT).

The expression

- must return a single value
- cannot be an array or return an array
- must not refer to any column which does not exist in the table

For example, suppose you want to create a view that displays the salary ranges for all jobs that pay at least \$60,000. The view, GOOD_JOB, based on the JOB table, selects the pertinent jobs and their salary ranges:

```
CREATE VIEW GOOD_JOB (JOB_TITLE, STRT_SALARY, TOP_SALARY) AS
    SELECT JOB_TITLE, MIN_SALARY, MAX_SALARY FROM JOB
        WHERE MIN_SALARY > 60000;
```

Suppose you want to create a view that assigns a hypothetical 10% salary increase to all employees in the company. The next example creates a view that displays all of the employees and their new salaries:

```
CREATE VIEW 10%_RAISE (EMPLOYEE, NEW_SALARY) AS
    SELECT EMP_NO, SALARY *1.1 FROM EMPLOYEE;
```

Read-only and updatable views

When a DML operation is performed on a view, the changes can be passed through to the underlying tables from which the view was created only if certain conditions are met. If a view meets these conditions, it is

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

updatable. If it does not meet these conditions, it is *read-only*, and writes to the view cannot be passed through to the underlying tables.

Values can only be inserted through a view for those columns named in the view. Firebird stores NULL values for unreferenced columns.

Naturally updatable views

A view is *naturally updatable* if all of the following conditions are met:

- It is a subset of a single table or another updatable view.
- All base table columns excluded from the view definition allow NULL values.
- The view's SELECT statement does not contain subqueries, a DISTINCT predicate, a HAVING clause, aggregate functions, joined tables, user-defined functions, or stored procedures.

If the view definition does not meet all of these conditions, it is considered read-only.

The following statement creates a naturally updatable view:

```
CREATE VIEW EMP_MNGRS (FIRST, LAST, SALARY) AS
    SELECT FIRST_NAME, LAST_NAME, SALARY
        FROM EMPLOYEE
    WHERE JOB_CODE = 'Mngr' ;
```

Special consideration for INSERT and UPDATE

On naturally updatable views, it is possible to include a parameter—WITH CHECK OPTION—which prevents a user or objects from inserting a new row, or modifying an existing row, which would cause the new record version to violate the view's selection conditions. For details, see [Using WITH CHECK OPTION](#) on page 372.

Changing the behavior of updatable views

Alternative behavior for naturally updatable views can be specified using triggers. Because Firebird does not, under some conditions, perform write-throughs on any view that has one or more triggers defined, it is possible to take complete control of what happens to any base table when users modify a view based on it.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

❖ If BEFORE triggers are defined for a naturally updatable view, it is part of its natural behavior to fire those triggers before executing the specified update operation (UPDATE, DELETE, or INSERT). Design and test with care any BEFORE triggers you define, to ensure that you do not get unplanned results as a result of double action.

For detailed information about creating triggers, see [Elements of procedures and triggers](#) on p. 495 of chapter 25.

Examples of read-only views

This statement uses a nested sub-select to create a view, making the view read-only:

```
CREATE VIEW ALL_MNGRS AS
    SELECT FIRST_NAME, LAST_NAME, JOB_COUNTRY FROM EMPLOYEE
        WHERE JOB_COUNTRY IN
            (SELECT JOB_COUNTRY FROM JOB
                WHERE JOB_TITLE = 'manager');
```

The next statement creates a view that joins two tables—it is also read-only:

```
CREATE VIEW PHONE_LIST AS
    SELECT EMP_NO, FIRST_NAME, LAST_NAME, PHONE_EXT, LOCATION, PHONE_NO
        FROM EMPLOYEE, DEPARTMENT
        WHERE EMPLOYEE.DEPT_NO = DEPARTMENT.DEPT_NO.
```

Making read-only views updatable

Views that are based on joins—including reflexive joins—and on aggregates cannot be updated directly. You can, however, write triggers that will perform the correct writes to the base tables when a DELETE, UPDATE, or INSERT is performed on the view. This Firebird feature can turn many non-updatable views into updatable views.

Not all views can be made updatable by defining triggers for them. For example, this read-only view attempts to count records from the client; but regardless of the triggers you define for it, all operations except SELECT always fail:

```
CREATE VIEW AS SELECT 1 FROM MyTable;
```

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

The following example creates two tables, creates a view that is a join of the two tables, and then creates three triggers—one each for DELETE, UPDATE, and INSERT—that will pass all updates on the view through to the underlying base tables.

```
CREATE TABLE Table1 (
    ColA INTEGER NOT NULL,
    ColB VARCHAR(20),
    CONSTRAINT pk_table PRIMARY KEY(ColA)
);

CREATE TABLE Table2 (
    ColA INTEGER NOT NULL,
    ColC VARCHAR(20),
    CONSTRAINT fk_table2 FOREIGN KEY REFERENCES Table1(ColA)
);

CREATE VIEW TableView AS
SELECT Table1.ColA, Table1.ColB, Table2.ColC
FROM Table1, Table2
WHERE Table1.ColA = Table2.ColA;

CREATE TRIGGER TableView_Delete FOR TableView BEFORE DELETE AS
BEGIN
    DELETE FROM Table1
    WHERE ColA = OLD.Cola;
    DELETE FROM Table2
    WHERE ColA = OLD.Cola;
END;
```

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

```
CREATE TRIGGER TableView_Update FOR TableView BEFORE UPDATE AS
BEGIN
  UPDATE Table1
  SET ColB = NEW.ColB
  WHERE ColA = OLD.ColA;
  UPDATE Table2
  SET ColC = NEW.ColC
  WHERE ColA = OLD.ColA;
END;
```

```
CREATE TRIGGER TableView_Insert FOR TableView BEFORE INSERT AS
BEGIN
  INSERT INTO Table1 values (NEW.ColA,NEW.ColB);
  INSERT INTO Table2 values (NEW.ColA,NEW.ColC);
END;
```

 For detailed information about creating triggers, see [Elements of procedures and triggers](#) on p. 495 of chapter 25.

Using WITH CHECK OPTION

WITH CHECK OPTION specifies rules for modifying data through views. This option can be included only if the views are naturally updatable.

A view created using WITH CHECK OPTION will require Firebird to verify that a row inserted or updated through a view satisfies its selection conditions—the WHERE clause of the SELECT portion of the CREATE VIEW statement—before allowing the operation to succeed. The effect is to prevent insertion or modification such that the new or changed row would not be visible to the view.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Examples

Suppose you want to create a view that allows access to information about all departments with budgets between \$10,000 and \$500,000. The view, SUB_DEPT, is defined as follows:

```
CREATE VIEW SUB_DEPT (DEPT_NAME, DEPT_NO, SUB_DEPT_NO, LOW_BUDGET) AS  
    SELECT DEPARTMENT, DEPT_NO, HEAD_DEPT, BUDGET  
        FROM DEPARTMENT WHERE BUDGET BETWEEN 10000 AND 500000  
    WITH CHECK OPTION;
```

The SUB_DEPT view references a single table, DEPARTMENT. If you are the creator of the view or have INSERT privileges, you can insert new data into the DEPARTMENT, DEPT_NO, HEAD_DEPT, and BUDGET columns of the base table, DEPARTMENT. WITH CHECK OPTION assures that all values entered through the view fall within the range prescribed for each column in the WHERE clause of the SUB_DEPT view.

The following statement inserts a new row for the Publications Department through the SUB_DEPT view:

```
INSERT INTO SUB_DEPT (DEPT_NAME, DEPT_NO, SUB_DEPT_NO, LOW_BUDGET)  
    VALUES ('Publications', '7735', '670', 250000);
```

Firebird inserts NULL values for all other columns in the DEPARTMENT base table that are not available directly through the view.

Modify a view definition?

The terms *updatable* and *read-only* refer to how the data in the underlying tables can be accessed, not to whether the *view definition* can be modified. Firebird does not provide an ALTER VIEW syntax.

- ❖ To modify a view definition, you must drop the view and then recreate it.



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

Dropping views

The DROP VIEW statement enables a view's owner to remove a view definition from the database. It does not affect the base tables associated with the view. You can drop a view only if:

- You are logged in as the user who created the view.
- The view is not used in another view, a stored procedure, or CHECK constraint definition. You must delete the associated database objects before dropping the view.

Syntax

The syntax for DROP VIEW is:

```
DROP VIEW name;
```

The following statement removes a view definition:

```
DROP VIEW SUB_DEPT;
```



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

CHAPTER 20

More About Databases

This chapter looks again at databases, discussing some of the techniques involved in maintaining and administering a Firebird database, including

- the capability of a dialect 3 Firebird database to be accessed as a read-only data store
- how to alter attributes of a database
- how to delete a database
- database shadowing
- an overview of garbage collection

Read-only databases

By default, databases are in read-write mode when created. Such files must be on a writable filesystem even if they are used only for SELECT, because Firebird writes information about transaction states to a data structure in the database file.

You have the option of changing a dialect 3 database to read-only mode. A read-only database has the following characteristics:

- Databases can be placed on CD-ROMs or in read-only filesystems as well as on read-write filesystems.
- Attempted INSERT, UPDATE, and DELETE operations on a read-only database generate an error.
 See the *Firebird Reference Guide*—[Error Codes and Messages](#) (ch. 8 p. 291).
- No metadata changes are possible.
- Generators in a read-only database do not increment and are allowed only to return the current value.
For example, in a read-only database, the following statement succeeds:

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

```
SELECT GEN_ID(generator_name, 0) FROM table_name;
```

The following statement fails with the error "attempted update on read-only database."

```
SELECT GEN_ID(generator_name, 1) FROM table_name;
```

- External files accessed through a read-only database open in read-only mode, regardless of the file's permissions at the file system level.
- The read-only feature requires that both client and database be dialect 3.

Making a database read-only

To switch the mode of a database between read-write and read-only, you must be either its owner or SYSDBA and you must have exclusive access to a database.

From within Firebird, you can change a read-write database to read-only mode in these ways:

- Use **gbak** to back up the database and restore it in read-only mode:

```
gbak -create -mode read_only old_database.gbk new_database.gdb
```

- Use **gfix** to change the mode to read-only:

```
gfix -mode read_only database.gdb
```

★ To set a database to read-only mode from any application that uses BDE, ODBC, or JDBC, use the *isc_service_start()* function with service action '*isc_action_svc_properties*'.

★ To distribute a read-write database on a CD-ROM, back it up and put the *database.gbk* file on the CD-ROM. As part of the installation, restore the database to the user's hard disk.

Read-only databases with InterBase® 5.x

- An InterBase® 5.x client can access a read-only database to perform SELECTs. No other operation succeeds.
- If a Firebird client tries to set an InterBase® 5.x to read-only mode, the server silently ignores the request. There is no way to make older databases read-only. You must upgrade them to Firebird.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Altering a database

The maximum size of the primary (or first) file of a Firebird database is determined by the limit which the operating system places on the size of a single file which can be simultaneously read from and written to by one or more users. When a database comes close to its maximum size, it is at risk of corruption. Firebird is designed to manage logically single databases that span multiple physical files in the host filesystem.

Unlike some other multi-file database management systems, Firebird does not require pre-emptive disk storage to be reserved when a database is created. Databases can be created with a minimal size and can be altered at any time to provide more capacity through the addition of *secondary files*.

Secondary files

Secondary files are useful for controlling the growth and location of a database. They permit database files to be spread across storage devices, although all files must be located on disks that are under the direct physical control of the Firebird server's host machine.



For more information on secondary files, see [Creating a multi-file database](#) on p. 278 of chapter 14.

The ALTER DATABASE statement

The purpose of ALTER DATABASE is to add one or more secondary files to an existing database. It requires exclusive access to the database.



For more information about exclusive database access, see [Getting exclusive access to a database](#) on p. 283 of chapter 14.

A database can be altered by its creator, the SYSDBA user or, on Linux/UNIX, any user with operating system root privileges.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Syntax

The syntax for ALTER DATABASE is:

```
ALTER {DATABASE | SCHEMA}
    ADD <add_clause>;
<add_clause> = FILE 'filespec' <fileinfo> [<add_clause>]
<fileinfo> = {LENGTH [=] int [PAGE[S]] | STARTING [AT [PAGE]] int }
[<fileinfo>]
```

You must specify a range of pages for each file either by providing the number of pages in each file, or by providing the starting page number for the file.

- ★ It is never necessary to specify a length for the last—or only—file, because Firebird always dynamically sizes the last file and will increase the file size as necessary until all the available space is used or until it reaches the maximum database file size, as determined by operating system limits on the size of shared access files.

The first example adds two secondary files to the currently connected database by specifying the starting page numbers:

```
ALTER DATABASE
    ADD FILE 'employee2.gdb' STARTING AT PAGE 10001
    ADD FILE 'employee3.gdb' STARTING AT PAGE 20001
```

The first secondary file will grow until it reaches 10000 pages, after which Firebird will begin storing new pages in the second file.

The next example does nearly the same thing as the previous example, but it specifies the secondary file length rather than the starting page number.

```
ALTER DATABASE
    ADD FILE 'employee2.gdb' LENGTH 10000
    ADD FILE 'employee3.gdb'
```

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

The effect is slightly different to the first example: in this case, Firebird will begin using the next file when it reaches the point where one more page will not fit into the previous file.

- ★ The difference has no effect on performance or overall database size.

Dropping a database

When a database is no longer required, it can be deleted from the server. A dropped database cannot be recovered, so

- 1 be certain that you really want it to be gone forever
- 2 take a backup of it if there is any chance that you will need anything from it in future

The command to delete a database is `DROP DATABASE`. It takes no parameters—you must be connected to it as its owner, as `SYSDBA` or (on Linux/UNIX) as a user with system root privileges.

The following statement deletes the primary file and all secondary files pertaining to the current database:

```
DROP DATABASE;
```

Database shadows

Firebird has a capability which can enable immediate recovery of a database in case of disk failure, network failure, or accidental deletion of the database. It is called *shadowing*. Shadowing has obvious benefits and some limitations, as well as administrative and system resource overheads.

A shadow is a maintained physical copy of a database which, when changes are written to the database, receives the same changes simultaneously. An active shadow thus always reflects the current state of the database.

The main tasks in setting up and maintaining shadowing are as follows:

- **CREATING A SHADOW** Shadowing begins with the creation of a *shadow*. For information about the different ways to define a shadow, see [Using CREATE SHADOW](#) on page 381.
- **DELETING A SHADOW** If shadowing is no longer desired, the shadow can be deleted. For more information about deleting a shadow, see [Dropping a shadow](#) on page 385.

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

- **ADDING FILES TO A SHADOW** A shadow can consist of more than one file. As shadows grow in size, files can be added to accommodate the increased space requirements.

Benefits of shadowing

Shadowing offers several benefits:

- Recovery is quick: Activating a shadow makes it available immediately.
- Creating a shadow does not require exclusive access to the database.
- You can control the allocation of disk space. A shadow can span multiple files on multiple disks.
- Shadowing does not use a separate process. The database process handles writing to the shadow.
- Shadowing runs behind the scenes and needs little or no maintenance.

Limitations of shadowing

Shadowing has the following limitations:

- Shadowing is useful only for recovery from hardware failures or accidental deletion of the database. User errors or software failures that corrupt the database are duplicated in the shadow.
- Recovery to a specific point in time is not possible. When a shadow is activated, it takes over as a duplicate of the database. Shadowing is an “all or nothing” recovery method.
- Shadowing can occur only to a local fixed disk. Firebird does not support shadowing to an NFS file system, mapped drive, tape, or other media.

Before creating a shadow

Before creating a shadow, consider the following questions:

- *Where will the shadow reside?*
A shadow should be created on a *different fixed disk* from where the main database resides. Because shadowing is intended as a recovery mechanism in case of disk failure, maintaining a database and its shadow on the same disk defeats the purpose of shadowing. The disk or disks should be physically attached to the host machine.
- *How will the shadow be distributed?*
A shadow can be created as a single disk file called a shadow file or as multiple files called a shadow set. To improve space allocation and disk I/O, each file in a shadow set can be placed on a different disk.

**Symbols** **Numerics** [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

- If something happens that makes a shadow unavailable, should users be allowed to access the database?

If a shadow becomes unavailable, Firebird can either deny user access until shadowing is resumed, or Firebird can allow access even though database changes are not being shadowed.

The mode (manual or automatic) in which the shadow is created affects the system's response to a shadow's becoming unavailable.

For more information about these modes, see [Auto mode and manual mode](#).

- If a shadow takes over for a database, should a new shadow be automatically created?

To ensure that a new shadow is automatically created, create a *conditional shadow*.

For more information, see [Conditional shadows](#).

Using CREATE SHADOW

Use the CREATE SHADOW statement to create a database shadow for the database to which you are currently connected. It does not require exclusive access and can be done without affecting other users.

A shadow can be created using a combination of the following options:

- Single-file or multifile shadows
- Auto or manual shadows
- Conditional shadows

The options are not mutually exclusive. For example, you can create a single-file, manual, conditional shadow.

Syntax

The syntax of CREATE SHADOW is:

```
CREATE SHADOW set_num [AUTO | MANUAL] [CONDITIONAL]
    'filespec' [LENGTH [=] int [PAGE[S]]] [<secondary_file>];
<secondary_file> = FILE 'filespec' [<fileinfo>] [<secondary_file>]
```



Symbols **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

```
<fileinfo> = {LENGTH[=]int [PAGE[S]] | STARTING [AT [PAGE]] int }
[<fileinfo>]
```

Creating a single-file shadow

To create a single-file shadow for the database **employee.gdb**, use the statement

```
CREATE SHADOW 1 'employee.shd';
```

The shadow is associated with the currently connected database, **employee.gdb**. The name of the shadow file is **employee.shd**, and it is identified by a shadow set number, 1, in this example. The shadow set number tells Firebird that all of the shadow files listed are grouped together under this identifier.

To verify that the shadow has been created, use the **isql** command **SHOW DATABASE**:

```
SHOW DATABASE;

Database: employee.gdb
Shadow 1: '/usr/firebird/employee.shd' auto
PAGE_SIZE 4096
Number of DB pages allocated = 392
Sweep interval = 20000
```

The page size of the shadow is the same as that of the database.

Shadow location

On **non-NFS systems**, which includes all Windows machines, the shadow must reside on the same host as the database. You cannot specify a different host name or a mapped drive as the location of the shadow.

On **Linux/UNIX systems**, it is possible to place the shadow on any NFS-mounted directory.

❖ You run the risk of losing the shadow if you experience problems with NFS, so this is not a recommended procedure.



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

Creating a multi-file shadow

The method for creating multi-file shadows is similar to the way you create multi-file databases. To create a multifile shadow, specify the name and size of each file in the shadow set. File specifications are platform-specific.

The following examples illustrate the creation of a multifile shadow on a Windows platform. They create the shadow files on the A, B, and C drives of the *FB_backup* server host.

The first example creates a shadow set consisting of three files. The primary file, **employee.shd**, is 10,000 database pages in length and the first secondary file is 20,000 database pages long. The final secondary file, as always, grows as needed.

```
CREATE SHADOW 1 'D:/shadows/employee.shd' LENGTH 10000
FILE 'D:/shadows/employee2.shd' LENGTH 5000
FILE 'D:/shadows/employee3.shd';
```

Instead of specifying the page length of secondary files, you can specify their starting pages. The previous example could be entered as follows:

```
CREATE SHADOW 1 'D:/shadows/employee.shd' LENGTH 10000
FILE 'D:/shadows/employee2.shd' STARTING AT 10000
FILE 'D:/shadows/employee3.shd' STARTING AT 30000;
```

In either case, you can use SHOW DATABASE to verify the file names, page lengths, and starting pages for the shadow just created:

```
SHOW DATABASE;
Database: employee.gdb
Owner: SYSDBA
Shadow 1: "D:\SHADOWS\EMPLOYEE.SHD" auto length 10000
          file D:\SHADOWS\EMPLOYEE2.SHD starting 10000
          file D:\SHADOWS\EMPLOYEE3.SHD starting 30000
PAGE_SIZE 1024
Number of DB pages allocated = 462
Sweep interval = 20000
```



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- ★ The page length allocated for secondary shadow files need not correspond to the page length of the database's secondary files. As the database grows and its first shadow file becomes full, updates to the database automatically overflow into the next shadow file.

Auto mode and manual mode

A shadow can become unavailable for the same reasons a database becomes unavailable: disk failure, network failure, or accidental deletion. If a shadow becomes unavailable, and it was created in AUTO mode, database operations continue without shadowing, without intervention from the SYSDBA. If the shadow was created in MANUAL mode, further access to the database is denied until the database administrator intervenes. The benefits of AUTO mode and MANUAL mode are compared in the following table:

TABLE 20-1 Auto vs. manual shadows

Mode	Advantage	Disadvantage
AUTO	Database operation is uninterrupted	Creates a temporary period when the database is not shadowed; the DBA might be unaware that the database is operating without a shadow.
MANUAL	Prevents the database from running unintentionally without a shadow	Halts database operation until the problem is fixed; needs intervention of the DBA

AUTO MODE

The AUTO keyword directs the CREATE SHADOW statement to create a shadow in AUTO mode:

```
CREATE SHADOW 1 AUTO 'employee.shd';
```

Auto mode is the default, so omitting the AUTO reserved word achieves the same result.

In AUTO mode, database operation continues even if the shadow becomes inoperable. Also,

- If the original shadow was created as a *conditional shadow*, a new shadow is automatically created.
- If the shadow was not conditional, a new shadow must be created manually.

 For more information about conditional shadows, see [Conditional shadows](#).



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

MANUAL MODE

The MANUAL reserved word directs the CREATE SHADOW statement to create a shadow in manual mode:

```
CREATE SHADOW 1 MANUAL 'employee.shd';
```

Manual mode is useful when continuous shadowing is more important than continuous operation of the database. When a manual-mode shadow becomes unavailable, further connections to the database are prevented. To allow database connections again, the database administrator must remove the old shadow file, delete references to it, and create a new shadow.

Conditional shadows

A shadow can be defined so that if it replaces a database, a new shadow will be automatically created, allowing shadowing to continue uninterrupted. A shadow defined with this behavior is called a *conditional shadow*.

To create a conditional shadow, specify the CONDITIONAL reserved word with the CREATE SHADOW statement. For example:

```
CREATE SHADOW 3 CONDITIONAL 'employee.shd';
```

Including the CONDITIONAL keyword directs Firebird to create a new shadow automatically when either of the following conditions occurs:

- The database or one of its shadow files becomes unavailable.
- The shadow takes over for the database following hardware failure.

Dropping a shadow

To stop shadowing, use the shadow number as an argument to the DROP SHADOW statement. DROP SHADOW deletes shadow references from a database's metadata, as well as the physical files on disk.

A shadow can be dropped by its creator, the SYSDBA user, or any user with operating system root privileges.

DROP SHADOW syntax

```
DROP SHADOW set_num;
```

The following example drops all of the files associated with the shadow set number 1:

```
DROP SHADOW 1;
```

If you need to look up the shadow number, use the **isql** command SHOW DATABASE.

```
SHOW DATABASE;
```

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

```
Database: employee.gdb
Shadow 1: 'employee.shd' auto
PAGE_SIZE 1024
Number of DB pages allocated = 392
Sweep interval = 20000
```

Increasing the size of a shadow

If a database is expected to increase in size, or if the database is already larger than the space available for a shadow on one disk, you might need to expand the size of the shadow.

To do this, drop the current shadow and create a new one containing additional files.

- To add a shadow file, first use `DROP SHADOW` to delete the existing shadow, then use `CREATE SHADOW` to recreate it with the desired number of secondary files.

★ The page length allocated for secondary shadow files need not correspond to the page length of the database's secondary files. As the database grows and its first shadow file becomes full, updates to the database automatically overflow into the next shadow file.

Database "hygiene"

Background garbage collection

Firebird uses a *multi-generational architecture*. This means that multiple versions of data rows are stored directly on the data pages. When a row is updated or deleted, Firebird keeps a copy of the old state of the record and creates a new version. This proliferation of *record back versions* can increase the size of a database.

To limit this growth, Firebird continually performs *garbage collection* in the background of normal database activity. Each time it gets clear time, where a particular row is clear of unresolved transactions, it gathers up back versions of that row, deletes them and frees up the database pages they occupied.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

The background garbage collection does nothing to obsolete row versions that are caught up in unresolved transactions—they will not be visited during normal housekeeping activity. To completely sanitize the database, Firebird can perform *database sweeping*.

Sweeping

Database sweeping is a systematic way of removing all outdated row versions and freeing the pages they occupied so that they can be reused. Periodic sweeping prevents a database from growing unnecessarily large. Not surprisingly, although sweeping occurs in an asynchronous background thread, it can impose some cost on system performance.

By default, a Firebird database performs a sweep when the *sweep interval* reaches 20,000 transactions. Sweeping behavior is configurable, however: it can be left to run automatically, the sweep interval can be altered, or automatic sweeping can be disabled, to be run manually on demand instead. Manual sweeping can be done from the command-line housekeeping program, `gfix`. Several other desktop tools are available that provide a GUI for performing sweeps.

 For more information about sweeping with `gfix`, see [Sweeping](#) on p. 485 of chapter 24.

Sweep interval

The Firebird server maintains an inventory of transactions. Any transaction that is uncommitted is known as an *interesting transaction*. The oldest of these “interesting” transactions (Oldest Interesting Transaction—OIT) marks the starting point for the sweep interval. If the sweep interval setting is greater than zero, Firebird initiates a full sweep of the database when the difference between the OIT and the newest transaction passes the threshold set by the sweep interval.

 For instructions, see [Setting the sweep interval](#) on p. 486 of chapter 24

Garbage collection during backup

Sweeping a database is not the only way to perform systematic garbage collection. Backing up a database achieves the same result, because the Firebird server must read every record, an action that forces garbage collection throughout the database. As a result, regularly backing up a database can reduce the need to sweep and helps to maintain better application performance.



Symbols [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

For more information about the benefits of backing up and restoring, see [Why back up and restore?](#) on p. 390 of chapter 21.

The database cache

The *database cache* contains all of the database pages (also called buffers) that are held in RAM at one time. *Database cache size* is the default number of database pages reserved for this cache. You can set the default database cache size at three levels:

- server level: applies to all databases
- database level: applies only to a single database
- connection level: applies only to a specific **isql** connection

❖ To reduce the likelihood of inappropriate database cache sizes, it is recommended that cache size be set at the database level rather than at the server level.

For a detailed discussion of this topic, see [Configuring the database cache](#) on p. 67 of chapter 5.

Forced writes

Forced Writes is synonymous with *synchronous writes*. The term "disabling Forced Writes" means switching the write behavior from synchronous to asynchronous.

- With forced writes enabled, new records, new record versions and deletions are physically written to disk immediately upon COMMIT.
- Asynchronous writes hold new and changed data in the file cache, relying on the flushing behavior of the operating system to make them permanent on disk.

On platforms that support asynchronous writes, Firebird is installed with forced writes enabled. Forced writes are not applicable to Windows 95/98 nor ME.

For discussion of the implications of disabling Forced Writes and instructions for setting it using **gfix**, see [Enabling and disabling Forced Writes](#) on p. 476 of chapter 24.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Validation and repair

In day-to-day operation, a database is sometimes subjected to events that pose minor problems to database structures. These events include:

- Abnormal termination of a database application. This does not affect the integrity of the database. When an application is canceled, committed data is preserved, and uncommitted changes are rolled back. If Firebird has already assigned a data page for the uncommitted changes, the page might be considered an orphan page. Orphan pages are unassigned disk space that should be returned to free space.
- Write errors in the operating system or hardware. These usually create a problem with database integrity. Write errors can cause data structures such as database pages and indexes to become broken or lost. These corrupt data structures can make committed data unrecoverable.

You should validate a database:

- Whenever a database backup is unsuccessful.
- Whenever an application receives a “corrupt database” error.
- Periodically, to monitor for corrupt data structures or misallocated space.
- Any time you suspect data corruption.

Firebird's validation toolset

The command-line tool for performing validation is **gfix**.



For full details on using **gfix** to perform database validation, see [Performing a database validation](#) on p. 481 of chapter 24.

Repairing a corrupt database

If you suspect you have a corrupt database, it is important to follow a proper sequence of recovery steps in order to avoid further corruption. For a detailed description of the recommended recovery procedure, see [Database repair How-to](#) on p. 481 of chapter 24.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

CHAPTER 21

Database Backup and Restore

A database *backup* saves the metadata and data of a database compactly to a file on a hard disk or other storage medium. It is strongly recommended that backup files be transferred to a portable storage medium and stored at a secure physical location remote from the server.

Why back up and restore?

Regular backup is essential for good housekeeping, data integrity, security and disaster fallback.

Firebird's backup and restore utility **gbak** creates and restores backups of your databases. It creates a platform-independent, stable snapshot of the database for archiving purposes which can be written to a disk file or to a named tape device.

You do not have to shut down the database to run a **gbak** backup—it can run whilst other users are using the database. However, any data changes that clients commit to the database after the backup begins are not recorded in the backup file.

- ❖ Operating systems usually include facilities to archive database files. Do not rely on backups of your Firebird databases made using these, or filesystem copy utilities or compression utilities like gzip or WinZip, *unless the database is completely shut down*. Such database copies are not "hygienic" - uncleared garbage will be restored along with the rest of the file.

gbak backup & restore tool

Firebird's backup utility **gbak** analyses and decomposes Firebird database files, storing cleaned metadata and data separately, in a compact format.

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

A backup made with **gbak** is not a database file and will not be recognized by the server. In order to become usable, it must be restored to a Firebird format that is readable by the server, using the version of **gbak** that corresponds to the version of Firebird server that is installed.

When restoring its files back to database format, performs validation of both metadata and data before using query-language commands internally to reconstruct the database and repopulate it with its data..

★ If corrupt structures or data are detected, **gbak** stops restoring and reports the condition. Its capabilities to analyze problems make it an invaluable helper when recovery of broken databases is being attempted.

gbak's other talents

gbak also performs an important set of other utility tasks in the course of analyzing, storing and restoring your database. Some are automatic; others can be requested by means of switches in the command-line call to **gbak**. The tasks include:

- garbage collection on outdated records
- balancing indexes, to refresh the performance capability of your database
- reclaiming space occupied by deleted records and packing the remaining data. This often reduces database size
- optionally changing the database page size
- optionally splitting the database into multiple files
- optionally distributing a multi-file database across multiple disks
- optionally upgrading an InterBase® database to Firebird
- optionally upgrading an InterBase® or Firebird database to a higher on-disk structure (ODS)
- optionally changing the database owner

Upgrading the on-disk structure

New major releases of the Firebird server often contain changes to the *on-disk structure* (ODS). If the ODS has changed and you want to take advantage of any new Firebird features, upgrade your databases to the new ODS.



Symbols **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

To upgrade existing databases to a new ODS, perform the following steps:

- Before installing the new ODS version of Firebird, back up databases using the old version of **gbak**
- Install the new version of the Firebird server as described in the chapter *Installing Firebird*.
- Once the new version is installed, restore the databases with the **gbak** that is installed along with it.
- The restored databases are able to use any new Firebird server features.

★ You need not upgrade databases to use a new version of Firebird. The new versions can still access databases created with a previous version, but cannot take advantage of any new Firebird features.

Database backup & restore rights

Only the database owner or the SYSDBA user can

- perform a backup
 - perform a restore that overwrites the database
- ★ The owner of a database is the user which created the database.

Any user can restore a database as long as the restored database will not overwrite an existing database.

Changing ownership of a database

A restored database file is owned by the user which performed the restore. This means that backing up and restoring a database is a mechanism for changing the ownership of a database.

❖ It also means that an unauthorized user can “steal” a database by restoring a backup file to a machine where he knows the SYSDBA password. It is important to ensure that your backup files are secured from unauthorized access.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

User name and password

When Firebird checks authority to run **gbak**, it determines the user according to the following hierarchy:

- 1 The -user that is specified, with a correct password, as switches in the **gbak** command
- 2 The user name and password specified in the **ISC_USER** and **ISC_PASSWORD** environment variables, provided they also exist in **isc4.gdb**
 - ❖ setting these environment variables is strongly *not* recommended, since it is extremely insecure.
- 3 Linux/UNIX only: If no user is specified at any of the previous levels, Firebird uses the UNIX login if the user is running on the server or on a trusted host.

Running a backup

Backing up to a single file

```
gbak [-B] [options] db_name target
```

The arguments and switches are described below.

Backing up a database to multiple files

When you are backing up a multifile database, specify *only the first file* in the backup command. You must not name the subsequent database files, or they will be interpreted as backup file names.

```
gbak [-B] [options] db_name target1 size1[k|m|g] target2 [size2[k|m|g]]  
target3
```

❖ Notice that a *size* parameter should be omitted from the last file in the multi-file sequence.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Arguments for gbak -B[backup]

db_name

is the path and file name of a database to back up. If backing up a multifile database, use only the name of the *first* (primary) database file.

target

is the path and file name of a storage device or backup file to which the backed up database is to be sent.

★ On UNIX, target can also be `stdout`. In this case, `gbak` writes its output to the standard output (usually a pipe)

size

is used only when backing up to multiple files. It is the length of a backup in bytes (default), Kilobytes (k), Megabytes (m) or Gigabytes (g). To specify one of the non-default units, use the single lower-case character, as indicated in the command template, above.

Switches for gbak -B[backup]

TABLE 21-1 `gbak` backup switches

Switch	Effect
-b[backup_database]	Causes <code>gbak</code> to run a back up the named database to file or device
-co[nvert]	Converts external files into internal tables
-e[xpand]	Creates the backup without compression
-fa[ctor] n	Uses a blocking factor <i>n</i> for a tape device
-g[arbage_collect]	Suppresses garbage collection during backup



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

TABLE 21-1 `gbak` backup switches

Switch	Effect
-i[gnore]	Causes checksums to be ignored during backup
-l[imbo]	(Lower-case letter “L”) causes limbo transactions to be ignored
-m[etadata]	Backs up just the metadata—no data are saved
-nt	Creates the backup in non-transportable format.
-ol[d_descriptions]	Backs up metadata in old-style (earlier InterBase®) format
-pa[ssword] password	Checks for password <i>password</i> before accessing the database
-role name	Connects under ROLE <i>name</i>
-se[rvice] servicename	Creates the backup files on the host where the original database files are located. See <i>Using gbak with Firebird Service Manager</i> . <i>servicename</i> invokes the Service Manager on the server host. Syntax varies with the network protocol in use: <ul style="list-style-type: none">• TCP/IP: <i>hostname:service_mgr</i>• Named pipes: \\<i>hostname\service_mgr</i>
-t[ransportable]	Default—creates a transportable backup
-u[ser] name	Checks for user name <i>name</i> before accessing a remote database
-v[erbose]	Shows what <code>gbak</code> is doing—provides running counts, etc.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

TABLE 21–1 `gbak` backup switches

Switch	Effect
-y [<i>filespec</i> suppress_output]	Directs status messages to <i>filespec</i> . Fails if the named file already exists. suppress_output suppresses output messages
-z	Shows the version numbers of both <code>gbak</code> and the Firebird engine.

Return value (Windows only)

A `gbak` backup returns an errorlevel of 0 on success, 1 on failure. If an error occurs, check the `interbase.log` file.

Target locations for backup files

If you run `gbak` from a remote machine:

- *without* the `-service` switch, it writes the backup files to the current directory or a specified location on the client machine, not on the server where the database resides. If you specify a location for the backup file, it must refer to the machine where `gbak` is executing. The location can be
 - on a drive that is local to the client machine
 - on a drive that the local machine is mapping to (Windows)
 - in a NFS location (Linux/UNIX)
- *with* the `-service` switch, the **Firebird Service Manager** on the server host will be invoked. The database and backup file specifications must then refer to a location and filename local to the server machine (not to the machine where `gbak` is executing).



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

Transportable backups

Accept the default `-transportable` switch if you operate in a multiplatform environment. This switch permits the database to be backed up in a format that can be read by the `gbak` program on a platform different to the one on which it was backed up.

- ❖ The `gbak` program on servers with lower on-disk structure (ODS) than your Firebird server and database will generally NOT be able to restore from the higher ODS backup. In practice, however, the InterBase® 5.x version of `gbak` appears to be capable of restoring most Dialect 1 databases.
- ❖ You should never attempt to back up databases with a version of `gbak` that does not match the server version on which the database runs.

Security considerations

It is a good precaution to set the read-only property on your backup files at the filesystem level after creating them, to prevent them from being accidentally overwritten.

You can protect your databases from being “kidnapped” on UNIX and on Windows NT/2000/XP systems by placing the backup files in directories with restricted access.

- ❖ Backup files that are kept on Windows 95/98/ME systems, or in unrestricted areas of other systems, are completely vulnerable.

Running a restore

Any Firebird user defined on the server can restore a database. In order to restore by overwriting an existing database, the user must be the SYSDBA user or the database owner.

- ❖ A backup should never be restored to overwrite a database that is in use. It is likely to cause corruption.



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

User-defined international objects

When restoring a backup file to a server other than the one on which it was backed up, you must ensure that any character sets and collations referenced in the backup file exist on the destination server.

Multiple-file restores

The `gbak` utility can restore single or multiple backup files to single or multiple database files.

Restoring to a single file

```
gbak { -C | -R } [options] source db_filename
```

Restoring to multiple files

```
gbak { -C | -R } [options] source db_filename1 size1 db_filename2 [size2  
db_filename3 ...]
```

Do not provide a file size for the last (or only) file of a restored database. Firebird always "grows" the last file as needed until all available space is used.

Restoring from multiple files

```
gbak { -C | -R } [options] source1 source2 [source3 ...] db_filename
```

When restoring from a multi-file backup, you must name all of the backup files, in the order in which they were backed up.

Restoring to multiple files from multiple files

```
gbak { -C | -R } [options] source1 source2 [source3 ...] db_filename1 size1  
db_filename2 [size2 db_filename3 ...]
```



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- ★ If you specify several target database files but have only a small amount of data, the target files are initially quite small—around 800KB for the first one and 4KB for subsequent files. They grow, in sequence, to the specified sizes as you populate the database.

Arguments for gbak -C[reate] and -R[estore]

The **-C[reate]** switch creates a new database from the backup. The **-R[estore]** switch will overwrite a database of the given db_filename provided the user is SYSDBA or the owner of the database.

- ★ the **-R[estore]** switch will also create a new database from the backup if the target database filespec is not found.

source

is the path and file name of a storage device or backup file from which to restore.

- ★ On UNIX, source can also be stdin, in which case gbak reads its input from the standard input (usually a pipe)

db_filename

is the path(s) and file name(s) of a database to restore to.

size

is used only when restoring to a multiple-file database. It is the length of a file in database pages. The minimum value is 200 database pages.

- ❖ Do not specify a size for the final database file or where restoring to a single-file database. The last file always expands as needed to fill all available space



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Switches for gbak -C[reate] and -R[estore]

TABLE 21–2 `gbak` restore switches

Switch	Effect
-C[reate_database]	Restores database to a new file
-b[uffers]	Sets default cache size (in database pages) for the restored database.
-I[nactive]	Makes indexes inactive in the restored database
-k[ill]	Suppresses creation of any shadows that were previously defined
-mo[de] {[read_write read_only]}	Specifies whether the restored database is to be writable. Possible values are <code>read_write</code> (default) and <code>read_only</code>
-n[o_validity]	Deletes validity constraints from restored metadata. Use if you need to restore data that you believe would not comply with validity constraints.
-o[ne_at_a_time]	Restores one table at a time. Can be used for partial recovery if database contains corrupt data.
-p[age_size] n	Resets page size to <i>n</i> bytes (1024, 2048, 4096, or 8192). Default is 4096. A page size of 16384 is possible if the filesystem supports 64-bit file I/O.
pas[sword] password	Checks for password <i>password</i> before attempting to (re)create the database
-r[eplace_database]	Restores database, replacing the existing file <i>db_filename</i> if it exists; if not, creates a new file.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

TABLE 21–2 `gbak` restore switches

Switch	Effect
-se[rvice] servicename	Creates the restored database on the host where the backup files are located. Use this switch if you are restoring to the same server that holds the backup file. It invokes the Firebird Service Manager on the server host and saves both time and network traffic. Syntax varies with the network protocol in use: <ul style="list-style-type: none">• TCP/IP: <i>hostname</i>:service_mgr• Named pipes: \\<i>hostname</i>\service_mgr• Local service mgr
-u[ser] name	Checks for user <i>name</i> before attempting to (re)create the database.
-use_[all_space]	Restores database with 100% fill ratio on every data page, instead of the default 80% fill ratio. To revert restored database to normal fill ratio, use the <code>gfix -use</code> switch, i.e. <code>gfix -use reserve</code>
-v[erbose]	Shows what <code>gbak</code> is doing—provides running counts, etc.
-y [filespec suppress_output]	<ul style="list-style-type: none">• If used with -v, directs status messages to <i>filespec</i>.• If used without -v and <i>filespec</i> is omitted, suppresses output messages
-z	Shows the version numbers of both <code>gbak</code> and the Firebird engine.

Return value (Windows only)

A `gbak` restore returns an errorlevel of 0 on success, 1 on failure. If an error occurs, check the `interbase.log` file.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Database size and performance

The size of a restored database is specified in database pages. The default size for database files is 200 pages. The default database page size is 4KB, so if the page size has not been changed, the default database size is 800KB. This is sufficient for only a very small database.

To change the size of the database pages, use the `-p[age_size]` switch when restoring. Changing the page size can improve performance in the following conditions:

- Storing and retrieving **blob data** is most efficient when the entire blob fits on a single database page. If an application stores many Blobs exceeding 4KB, a larger page size reduces the time for accessing blob data.
- Firebird performs better if rows do not span pages. Consider increasing the page size if a database contains any tables with **long rows of data**.
- If a database contains any **large indexes**, a larger database page size reduces the number of levels in the index tree. The smaller the depth of indexes, the faster they can be traversed. Consider increasing the page size if index depth is greater than three on any frequently used index.
- **Reducing the page size** may be appropriate if most transactions involve only a few rows of data, because the volume of data needing to be passed back and forth is lower and less memory is used by the disk cache.

Using gbak with Firebird Service Manager

The `-service` switch saves a significant amount of time and network traffic when you want the backup to be created on the host on which the database resides.

- On a Windows server, the `-service` switch offers no advantage if the connection is local
- On a Linux/UNIX server, it saves time and traffic even on a localhost connection

When you run `gbak` with the `-service` switch, it operates differently from the way it does otherwise:

- The `-service` switch causes `gbak` to invoke the backup and restore functions of Firebird's Service Manager *on the server where the database resides*.

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

- Without the -service switch, `gbak` executes on the machine where it is invoked—typically a client—and writes the backup file to a location on or referenced by that machine.

You have the option to specify another target machine when using the -service switch, but the benefits of reduced time and network traffic are lost.

You can back up to multiple files and restore from multiple files using Service Manager.

When you use the -service switch, you specify the host name followed by the constant string "service_mgr". The syntax varies according to the network protocol you are using.

Backing up with Service Manager

```
gbak -b [options] -se[rvice] host_service db_name filespec
```

Restoring with Service Manager

```
gbak {-c|-r} [options] -se[rvice] host_service filespec db_name
```

Arguments for the -service switch

host_service

Consists of the protocol-specific host (server) name plus the constant `service_mgr`:

TABLE 21-3 Syntaxes for host_service

Protocol	Syntax
TCP/IP	<code>hostname:service_mgr</code>
Named pipes	<code>\\\hostname\service_mgr</code>
Local (Windows only)	<code>service_mgr</code>



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Restoring on Linux/UNIX

To restore a database that has been backed up using the Service Manager, you must either

- use the Service Manager for the restore; *or*
- be logged onto the system as the user which was current when the backup was created (either *root*, *firebird* or *interbase*). This restriction arises because that user is the owner of the backup file at the filesystem level when the Service Manager is invoked, causing it to be readable to only that user.

★ On Windows platforms, the system-level constraints do not apply.

When **gbak** is used to back up a database without the **-service** option, the filesystem owner of the backup file is attributed to the login of the person who ran **gbak**.

Backup and restore examples

★ The examples in this section use forward slashes exclusively. Firebird accepts either forward or backward slashes for paths on Windows platforms.

Backup

The following example backs up **magic.gdb**, which resides on the server **phoenix** and writes the backup file to the current directory of the client machine where **gbak** is running. **magic.gdb** can be either a single-file database or the name of the first file in a multifile database.

```
gbak -b -user gandalf -pas baggins phoenix:/magic.gdb magic.gbk
```

The next example backs up **magic.gdb**, which resides on the server **phoenix** and writes the backup file to the **C:/backups** directory on the client machine where **gbak** is running. As before, **magic.gdb** can be a single file database or the name of the first file in a multifile database.

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

```
gbak -b -user gandalf -pas baggins phoenix:/magic.gdb  
C:/backups/magic.gbk
```

- ❖ Using either of these syntaxes (without the -se switch) copies a lot of data over the net.

The next example backs up the same database on **phoenix**, but uses the **-se[rvice]** switch to invoke the Service Manager on **phoenix**, which writes the backup to the **\january** directory on **phoenix**. Notice that the syntax (**phoenix:service_mgr**) indicates a TCP/IP connection:

```
gbak -b -user gandalf -pas baggins -se phoenix:service_mgr /magic.gdb  
/january/magic.gbk
```

- ★ This command causes very little network traffic and is therefore faster than performing the same task without the -service switch.

The next example again backs up **magic.gdb** on server **phoenix** to multiple files in the **/backups** directory on **phoenix** using the Service Manager. This syntax backs up a single file or multifile database and uses a minimum of time and network traffic. It converts external files as internal tables and creates a backup in a transportable format that can be restored on any Firebird-supported platform.

To back up a multifile database, name only the first file in the backup command. In this example, the first two backup files are limited to 500K. The last one expands as necessary.

```
gbak -b -user gandalf -pas baggins -co -t -se phoenix:service_mgr  
/magic1.gdb /backups/backup1.gbk 500k /backups/backup2.gbk 500k  
/backups/lastBackup.gbk
```

Restore examples

The first example restores a database that resides in the **/backups** directory on the machine where **gbak** is running and restores it to **phoenix**, overwriting an existing (but inactive) database.



Symbols **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

```
gbak -r -user frodo -pas pipeweek C:\backups\magic.gbk phoenix:/foo.gdb
```

The next example restores a multifile database from the */january* directory of **phoenix** to the */currentdb* directory of **phoenix**. This command runs on the server by invoking Service Manager, thus saving time and network traffic. In this example, the first two files of the restored database are 500 pages long and the last file grows as needed.

```
gbak -r -user frodo -pas pipeweek -se phoenix:service_mgr  
/january/magic1.gbk /january/magic2.gbk /january/magicLast.gbk  
/currentdb/magic1.gdb 500 /currentdb/magic2.gdb 500  
/currentdb/magicLast.gdb
```

The next example executes on server **phoenix** using Service Manager and restores a backup that is on **phoenix** to another server called **icarus**.

```
gbak -r -user frodo -pas pipeweek -se phoenix:service_mgr  
/january/magic.gbk icarus:/currentdb/magic.gdb
```

gbak Error messages

TABLE 21-4 error messages from **gbak** backup and restore

Error Message	Causes and Suggested Actions to Take
Array dimension for column <string> is invalid	Fix the array definition before backing up
Bad attribute for RDB\$CHARACTER_SETS	An incompatible character set is in use
Bad attribute for RDB\$COLLATIONS	Fix the attribute in the named system table



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

TABLE 21–4 error messages from **gbak** backup and restore

Error Message	Causes and Suggested Actions to Take
Bad attribute for table constraint	Check integrity constraints; if restoring, consider using the -no_validity option to delete validity constraints
Blocking factor parameter missing	Supply a numeric argument for “factor” option
Cannot commit files	Database contains corruption or metadata violates integrity constraints
	Try restoring tables using -one_at_a_time option, or delete validity constraints using -no_validity option
Cannot commit index <string>	Data might conflict with defined indexes. Try restoring using “inactive” option to prevent rebuilding indexes
Cannot find column for blob	..
Cannot find table <string>	..
Cannot open backup file <string>	Correct the file name you supplied and try again
Cannot open status and error output file <string>	Messages are being redirected to invalid file name. Check format of file or access permissions on the directory of output file
Commit failed on table <string>	Data corruption or violation of integrity constraint in the specified table. Check metadata or restore “one table at a time”



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

TABLE 21–4 error messages from **gbak** backup and restore

Error Message	Causes and Suggested Actions to Take
Conflicting switches for backup/restore	A backup-only option and restore-only option were used in the same operation; fix the command and execute again
Could not open file name <string>	Fix the file name and re-execute command
Could not read from file <string>	Fix the file name and re-execute command
Could not write to file <string>	Fix the file name and re-execute command
Datatype n not understood	An illegal datatype is being specified
Database format n is too old to restore to	The gbak version used is incompatible with the Firebird version of the database. Try backing up the database using the -expand or -old options and then restoring it
Database <string> already exists. To replace it, use the –R switch	You used -create in restoring a back up file, but the target database already exists. Either rename the target database or use -replace
Could not drop database <string> (database might be in use)	You used -replace in restoring a file to an existing database, but the database is in use. Either rename the target database or wait until it is not in use
Do not recognize record type n	..
Do not recognize <string> attribute n -- continuing	..
Do not understand BLOB INFO item n	..



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

TABLE 21–4 error messages from **gbak** backup and restore

Error Message	Causes and Suggested Actions to Take
Error accessing BLOB column <string> -- continuing	..
ERROR: Backup incomplete	The backup cannot be written to the target device or file system. Cause could be insufficient space, a hardware write problem, or data corruption
Error committing metadata for table <string>	The table could be corrupt. If restoring a database, try using -one_at_a_time to isolate the table
Exiting before completion due to errors	This message accompanies other error messages and indicates that back up or restore could not execute. Check other error messages for the cause.
Expected array dimension n but instead found m	Try redefining the problem array
Expected array version number n but instead found m	Try redefining the problem array
Expected backup database <string>, found <string>	Check the name of the backup file being restored
Expected backup description record	...
Expected backup start time <string>, found <string>	..
Expected backup version 1, 2, or 3. Found n	..
Expected blocking factor, encountered <string>	The -factor option requires a numeric argument

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**TABLE 21–4 error messages from **gbak** backup and restore

Error Message	Causes and Suggested Actions to Take
Expected data attribute	..
Expected database description record	..
Expected number of bytes to be skipped, encountered <string>	...
Expected page size, encountered <string>	The -page_size option requires a numeric argument
Expected record length	..
Expected volume number n, found volume n	When backing up or restoring with multiple tapes, be sure to specify the correct volume number
Expected XDR record length	..
Failed in put_blr_gen_id	..
Failed in store_blr_gen_id	...
Failed to create database <string>	The target database specified is invalid; it might already exist
Column <string> used in index <string> seems to have vanished	An index references a non-existent column. Check either the index definition or column definition
Found unknown switch	An unrecognized gbak option was specified
Index <string> omitted because n of the expected m keys were found	..
Input and output have the same name	Disallowed. A backup file and database must have unique names; correct the names and try again



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

TABLE 21–4 error messages from **gbak** backup and restore

Error Message	Causes and Suggested Actions to Take
Length given for initial file (n) is less than minimum (m)	Insufficient space was allocated for restoring a database into multiple files. Firebird automatically increases the page length to the minimum value. No action necessary
Missing parameter for the number of bytes to be skipped	..
Multiple sources or destinations specified	Only one device name can be specified as a source or target
No table name for data	The database contains data that are not assigned to any table. Use gfix to validate or mend the database
Page size is allowed only on restore or create	The -page_size option was used during a back up instead of a restore
Page size parameter missing	
The -page_size option requires a numeric argument	Page size specified (n bytes) rounded up to m bytes
Invalid page sizes are rounded up to 1024, 2048, 4096, 8192 or 16384, whichever is closest	Page size specified (n) greater than limit (8192 bytes). Specify a page size of 1024, 2048, 4096, or 8192
Password parameter missing	The back up or restore is accessing a remote machine. Use the -password switch and specify a password
Protection is not there yet	Unimplemented option -unprotected used

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**TABLE 21–4 error messages from **gbak** backup and restore

Error Message	Causes and Suggested Actions to Take
Redirect location for output is not specified	You specified an option reserved for future use by Firebird
REPLACE specified, but the first file <string> is a database	Check that the file name following the -replace option is a backup file rather than a database
Requires both input and output file names	Specify both a source and target when backing up or restoring
RESTORE: decompression length error	Possible incompatibility in the gbak version used for backing up and the gbak version used for restoring. Check whether -expand should be specified during back up
Restore failed for record in table <string>	Possible data corruption in the named table
Skipped n bytes after reading a bad attribute n	..
Skipped n bytes looking for next valid attribute, encountered attribute m	..
Trigger <string> is invalid	..
Unexpected end of file on backup file	Restoration of the backup file failed; the backup procedure that created the backup file might have terminated abnormally. If possible, create a new backup file and use it to restore the database
Unexpected I/O error while <string> backup file	A disk error or other hardware error might have occurred during a backup or restore
Unknown switch <string>	An unrecognized gbak option was specified



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

TABLE 21–4 error messages from **gbak** backup and restore

Error Message	Causes and Suggested Actions to Take
User name parameter missing	The backup or restore is accessing a remote machine. Supply a user name with the -user switch
Validation error on column in table <string>	The database cannot be restored because it contains data that violates integrity constraints. Try deleting constraints from the metadata by specifying -no_validity during restore
Warning -- record could not be restored	Possible corruption of the named data
Wrong length record, expected n encountered n

[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

CHAPTER 22

Managing Security

Firebird has no facility to encrypt or decrypt data (other than user passwords) which pass through its client interface. It does provide certain restrictions on the use of Firebird tools to limit access to databases but, ultimately, at the software level alone, no system is safe from a raider who is determined to access your databases without authority.

This chapter draws attention to some areas where you need to take precautions with your Firebird server setup and examines Firebird's provisions for maintaining access control at both server and database levels.

Filesystem Security

Windows NT/2000 or XP

On Windows NT/2000 or XP systems, services run under the system Administrator account. Unless you have overriding reasons to do otherwise, you should run the Firebird server as a service and you are strongly recommended to protect a database file from unauthorized file-level access by making it readable only by the system Administrator account.

- ❖ When read access is thus restricted, remote clients must connect to the server through the TCP/IP protocol, not through Windows Networking (NetBEUI).

Windows 95/98 and ME

When security is an issue, Windows 95/98 and ME systems are unsuitable as Firebird host servers because they lack support for file-level security. Anyone with access to the filesystem can readily duplicate the database with the `copy` command or with a Windows GUI equivalent (drag 'n' drop, copy/paste, etc.).



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

False assumptions

- A file-copy is bound to be corrupt, so it is no use to an intruder.

Do not assume that a file copy of a running database will be corrupt and unusable. An illicit copy may well prove to be usable, especially if update activity on the database is relatively infrequent, or if the attacker can try copying repeatedly.

- *gbak* files are not databases so they are no use to an intruder.

Anybody can recreate your database from an unauthorized copy of your *gbak* file. Do not store backups on Windows 95/98 or ME systems.

Linux/UNIX

On Linux/UNIX, only the *root*, *firebird* or *interbas(e)* users may start the Firebird service. However, it is strongly recommended that you do not log on as *root* to start the server. If *root* owns the service, it increases vulnerability to accidental or malicious compromising of the Firebird server.

On these platforms it is not necessary to store backup files in directories with restricted privileges. It is sufficient to restrict permissions on the files.

Users other than the responsible owner and group can see a backup file but they cannot access it, provided all of the following are true:

- the backup file has permission 600 (`rw-----`) or 640 (`rw-r----`)
- only trusted persons belong to the groups
- the directory has permission `rwxr--r--`

Of course, the directory should not be writable by unauthorized users, since that would give the wrong people privileges to delete the file.

★ If the user or backup script issues the command `umask 077` (or `027`, as appropriate) before running *gbak*, unauthorized persons will not be able to access the backup file, no matter what permissions are set on the directory.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

External Objects

Firebird has two features that extend the capabilities of the database and provide flexibility not featured in most other database systems. They are

- user-defined functions (UDFs)
- external files accessible as database tables

Useful as they are, these two features require some care to avoid security risks.

User-defined functions

UDFs exist in shared object libraries outside the Firebird server engine. Once declared to a database and called in a SQL statement, a UDF runs in the same address space and with the same privileges as the server process. A "bad" UDF can crash the server. A malicious one could wreak far-ranging damage to your database.

Accordingly, take especial care that

- any UDFs are exhaustively tested and installed by trusted users or administrators, before being allowed to run in your Firebird production server's space.
- UDF libraries are never installed in directories to which unauthorized users can gain write access. Ensure that the `../Firebird/UDF` directory, or any other directories configured in a SuperServer installation as `external_function_directory`, are appropriately privileged to make them secure.

External files

By means of the CREATE EXTERNAL TABLE command, any Firebird schema can map a database table to a specified filesystem file and operate on fixed-format records stored in such files. The database engine itself has no innate capability to determine the integrity of data being accessed or exported from the database by this technique. Without careful handling, external files can make the database vulnerable to both accidental and intentional data corruption.

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

Accordingly, it is important to protect the database from the possibility of accessing external files that have been written by unauthorized users. In Firebird, you can configure a specific directory to store externally mapped files. It is strongly recommended that you create one or more suitably privileged directories in the server's physical domain for just this purpose.

- In the Firebird configuration file on the server (**ibconfig** on Windows, **isc_config** on Linux/UNIX) make an entry for each directory where you want the server to search for external files.
★ Notice that you must enclose the path in double quotes.

There is no limit to the number of directories that can be searched. Firebird treats a series of entries as a list and will search through them in order specified in the configuration file.

On Windows

```
EXTERNAL_FILE_DIRECTORY "H:\external1"  
EXTERNAL_FILE_DIRECTORY "X:\external2"
```

On Linux/UNIX

```
EXTERNAL_FILE_DIRECTORY "/usr/external1"
```

- Where applications are writing data to files intended for use by the database as external files, you must take precautions to prevent unauthorized access to client software.

Password protection

isc4.gdb and the masterkey password

The Firebird server provides user access screening by means of its security database, **isc4.gdb**, which is installed in the Firebird root directory. Unfortunately, in Firebird versions up to and including release 1, the name and location of the security database are hard-coded.

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

The **isc4.gdb** database stores definitions of all users which have access to the Firebird server. A password must be defined for each user. Passwords are case-sensitive.

Currently, only the first eight characters of a password are significant. Hence, for example, the passwords *masterkey* and *masterkeeper* are seen by the server as identical.

All new installations of Firebird install the SYSDBA user into **isc4.gdb** with the password *masterkey*. Obviously, this is widely known and is not intended to be secure. It should be changed at the first opportunity.

❖ Because it is impossible to retrieve lost passwords by querying the system, you should have a bomb-proof system for recording user passwords whenever they are changed.

Vulnerability of **isc4.gdb**

It is possible for someone with filesystem level access to **isc4.gdb** to overwrite it, either by a filesystem copy or by restoring a **gbak** backup over the current one while logged in as SYSDBA. If the copy of the unmodified **isc4.gdb** from the original is used, SYSDBA access to the server and databases becomes available, with the *masterkey* password.

Whilst this is a good way to retrieve database access when passwords have been lost, it is also a great way to access a database that was stolen from an unsecured server machine. The only way to make certain that your database cannot be stolen in this manner is to house the server in a location to which unauthorized access is impossible and to eliminate any possibility of unauthorized access at filesystem level.

- Password insecurity is the most frequent cause of security breaches. The 8-character limit on Firebird passwords makes it relatively easy to guess obvious passwords like *password* or *user*, personal names or job titles. Insist on obscure passwords and keep the hardcopy password list in a secure place.
- Ensure that **isc4.gdb** is visible only to appropriate users and put a procedure in place to monitor and lock out repeated attempts to log on.
- Because the security database is simply another database on the Firebird server, it is possible for SYSDBA to grant privileges on its tables to other users. Before you do this, make sure you

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

understand every way in which you can introduce security risks and be quite certain that you have all the loopholes blocked.

- Don't host sensitive databases on Windows operating systems that do not provide filesystem security. This means Windows 95/98, ME and possibly the Home Edition of Windows XP.



Firebird comes with a command-line tool, `gsec`, which only SYSDBA may use, to add, modify and delete users and their passwords in `isc4.gdb`. It is fully documented later in this chapter, in the section [Maintaining the security database](#).

Owner of the server process on Linux/UNIX

Some older default installations for Linux and possibly some UNIX platforms may have installed the Firebird server to be owned and run by `root` user. This is undesirable from the point of view of security. A script file (`/opt/firebird/bin/CSchangeRunUser.sh`) is provided to enable the runtime user to be changed where this problem exists.

Web and other n-tier server applications

Relying on default user names can have unforeseen effects, including unintentionally bestowing the privileges of the database owner, or even the server process owner, on ordinary users. It is strongly recommended that you have your application enforce input of a user name and password before any calls are made to the Firebird server process.

Network Security

Sniffers

Much of the communication between the client and the server carries sensitive information which can be quite easily "sniffed" by someone eavesdropping network communications. For example, the encrypted password could be sniffed and used to gain unauthorized access to the server.

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

It is vital, therefore, to ensure that all pieces of the network path between any client and the server are trusted. Add-on products can be purchased which provide encrypted network tunnelling solutions to block potentially insecure pathways.

Firewalls

Placing your server machines behind a firewall is recommended, for obvious reasons.

It may be less obvious that providing *firewall protection to client processes* is also a good idea. It is possible for a rogue user running on a trusted client machine to feed incorrect information to the server and gain privileged access to its databases. Windows clients are notoriously insecure.

A Linux/UNIX server can be configured to recognize trusted clients explicitly. From there, the server implicitly trusts a process running on a trusted client.

Denial-of-Service Attacks

The Firebird code has a large number of string copy commands that do not check the length of the data they are requested to copy. Certain of these overruns may be able to be manipulated externally by passing large strings of binary data into SQL statements or pushing random garbage into the server port (currently 3050). These days, use of these functions is a common technique for malicious buffer overrun attacks intended to bring down servers.

These vulnerabilities are more easily exploited if the server and client processes are not running on trusted networks and/or are not protected by firewalls.

Defensive programming can help to pre-empt D-o-S attacks on your system. Validating the lengths of input strings on web data entry pages, for example, may be extremely useful.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Restrictions on use of Firebird tools

User restrictions

Firebird command-line utilities that perform security-sensitive operations restrict the users who are permitted to use them to SYSDBA and, in the case of database-specific operations, the owner of the database. The affected utilities are **gfix**, **gstat** and the **-b(backup)** and **-r(replace)** options of **gbak**.

Anyone can use the **gbak -c[reate]** option to recreate a database from a backup file. For security reasons, make sure your backup files are stored in a secure location to prevent unauthorized persons from restoring databases and thus gaining access to them.

Running gstat on Linux/UNIX

On Linux/UNIX platforms, a user needs filesystem read access to the database file in order to run the database statistics utility **gstat**. If SYSDBA or the database owner is not logged on as the *root*, *firebird* or *interbas(e)* user, the permissions on the database file must be changed to include read permission for the affected group.

Maintaining the security database

Users are always required to log on to a Firebird database by supplying a valid user name and password in the connection string. User names and passwords of users which are "known" to the Firebird server are stored in a table called **USERS** in the security database **isc4.gdb**. This database is located in your Firebird root directory and should not be moved from there.

The gsec utility

Firebird provides the **gsec** utility as a command-line interface for maintaining the security database.

★ Only the SYSDBA user can run **gsec**.

You can run **gsec** interactively or you can call **gsec** commands directly from the operating system shell.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- ★ If the ISC_USER and ISC_PASSWORD environment variables for SYSDBA are exposed and you are logged in as *root* on Linux/UNIX or as *Administrator* on Windows NT/2000, you can invoke **gsec** without entering -user and -password parameters.

Starting a gsec interactive session

At the command line, type

```
gsec -user sysdba -password masterkey
```

The prompt changes to **GSEC>**, indicating that you are in interactive mode.

To quit an interactive session, type **QUIT** at the prompt.

Running gsec remotely

SYSDBA can use **gsec** on a client machine to administer users in a security database on a remote server. To connect to a remote **isc4.gdb**, use the **-database** switch, along with the remote database server and path specification. For example:

```
gsec -database jupiter:/usr/firebird/isc4.gdb -user sysdba -password  
masterkey
```

Passwords are case-sensitive and only the first eight characters are significant. It is strongly recommended that you choose passwords that are obscure and that you keep records of passwords and changes off-line.

User names are case-insensitive and are stored in upper case, regardless of how they are entered.

Using gsec interactive commands

gsec provides a number of commands, along with optional parameter switches corresponding to columns in the **USERS** table. Typically, a command can be accompanied by one or more parameter switches to enable the SYSDBA to maintain **USERS** records without need for direct SQL statements.

The supported command switches are **display**, **add**, **modify**, **delete**, **help** (or **?**) and **quit**.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- ★ The argument **username**, for the USERS record which is the target of the operation, is obligatory for **add**, **modify** and **delete**.

Parameter switches

One or more of the following optional parameter switches may be included as additional arguments to the **add** and **modify** command lines when inserting or editing records:

:

TABLE 22-1 gsec USER parameter switches

Switch	Description
-database <i>filespec</i>	Needed when running gsec from a remote workstation. <i>filespec</i> is the fully-qualified server and file path of the primary database file.
-pw	Password string to be stored for the target USER. Required for the -add switch.
-uid <i>integer</i>	User ID of the target USER, an optional identifying code number. An unassigned UID defaults to 0.
-gid <i>integer</i>	Group ID for target USER. An unassigned GID defaults to 0.
-fname <i>string</i>	First Name of target USER.
-mname <i>string</i>	Middle Name of target USER.
-lname <i>string</i>	Last Name of target USER.

display

Displays the main columns of the USERS table in **isc4.gdb**. Passwords are never displayed.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

GSEC> display

user name	uid	gid	full name
FRED	123	345	Fred Flintstone
BARNEY	124	345	Barney Rubble
BETTY	125	345	Betty Rubble
...			

To display the same information for a single row from the USERS table:

GSEC> display *username*

e.g.

GSEC> display mmouse

user name	uid	gid	full name
MMOUSE	25		MICHAEL MOUSE

a[dd]

Adds a user to the USERS table.

a[dd] *username* -pw *password* [*switches*]

where *username* is a unique, new user name and *password* is the password associated with that user.

❖ The abbreviation for the -password switch when adding a user or changing a password is **-pw**. Do not confuse this with the SYSDBA log-on password switch, which is **-pa**.

For example:

To add user *mmouse* and assign the password *veritas*, enter:

GSEC> add mmouse -pw veritas

To add authorization for a user named Donald Duck with user name *dduck* and password *whatsup*, enter:

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

```
GSEC> add dduck -pw whatsup -fname donald -lname duck
```

To verify the new entry, use the **display username** command:

```
GSEC> display dduck
```

user name	uid	gid	full name
DDUCK	0	0	DONALD DUCK

mo[dy]

For changing (editing) a column value on an existing USERS record. Supply the user name for the entry to change, followed by one or more switches indicating the items to change and the new value for each.

For example, to set the user ID of user *mmouse* to 25, change the first name to Michael and change his password to *icecream*, enter the following switches:

```
GSEC> modify mmouse -uid 25 -fname Michael -pw icecream
```

To verify the changed line, use **display** followed by the user name:

```
GSEC> display mmouse
```

user name	uid	gid	full name
MMOUSE	25	0	MICHAEL MOUSE

★ To modify a user name, first use the **delete** command to delete the old entry in *isc4.gdb*, then enter the new user name along with the other information.

de[lete]

```
de[lete] username
```

Deletes user *username* from the **USERS** table. This command takes just this single argument. For example:



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

GSEC> delete mmouse

You can use the display command to confirm that the entry has been deleted.

h[elp] or ?

Either of these switches displays a summary of the **gsec** commands, switches and syntax.

q[uit]

Ends the interactive session.

Using gsec from a command prompt

To use **gsec** from a command prompt, precede each command with **gsec** and convert each **gsec** command to a command switch by prefixing it with a hyphen (-).

For example, to add user "aladdin" and assign the password, "sesame", you would enter the following on the command line:

for Windows:

C:> gsec -add aladdin -pw sesame

for Linux/UNIX:

firebird> gsec -add aladdin -pw sesame

-display

To display the contents of the **USERS** table, enter:

C:> gsec -display

..and so on.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
gsec error messages

TABLE 22-2 gsec (user security) error messages

Error Message	Causes and Suggested Actions to Take
Add record error	The add command either specified an existing user, used invalid syntax, or was issued without appropriate privilege to run gsec . Change the user name or use modify on the existing user.
<string> already specified	During an add or modify , you specified data for the same column more than once. Retype the command.
Ambiguous switch specified	A command did not uniquely specify a valid operation.
Delete record error	The delete command was not allowed. Check that you have appropriate privileges to use gsec .
Error in switch specifications	This message accompanies other error messages and indicates that invalid syntax was used. Check other error messages for the cause.
Find/delete record error	Either the delete command could not find a specified user, or you do not have appropriate privilege to use gsec .
Find/display record error	Either the display command could not find a specified user, or you do not have appropriate privilege to use gsec .
Find/modify record error	Either the modify command could not find a specified user, or you do not have appropriate privilege to use gsec .
Incompatible switches specified	For example, you entered multiple switches for a delete command, which requires only the mandatory <i>username</i> argument. Correct the syntax and try again.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

TABLE 22–2 gsec (user security) error messages

Error Message	Causes and Suggested Actions to Take
Invalid parameter, no switch defined	You specified a value without a preceding argument.
Invalid switch specified	You specified an unrecognized option. Fix it and try again.
Modify record error	Invalid syntax for modify command. Fix it and try again. Also check that you have appropriate privileges to run gsec .
No user name specified	Specify a user name after add , modify , or delete .
Record not found for user: < <i>string</i> >	An entry for the specified user could not be found. Use display to list all users, then try again.
Unable to open database	The isc4.gdb security database does not exist or cannot be located by the operating system.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Database-level security

Default security and access

Owner

A database and all its objects (tables, views and stored procedures) are secured against unauthorized access when they are created. Initially, only a table's creator, its *owner*, has access to an object (table, view or stored procedure) and only its owner can permit other users to access it.

SYSDBA and 'Superuser'

The SYSDBA user has special rights to all databases and the objects within them, regardless of which user owns them. Furthermore, on operating systems that implement the concept of a superuser—a user with *root* or *locksmith* privileges—such a user also has full access and destructive rights access to all databases and their objects.

SQL access privileges

Access by any other users but the owner, the Firebird SYSDBA or the operating system Superuser is controlled by granting sets of rights—permissions to perform certain operations—each right linking one user to one object in one database.

Firebird's SQL language provides a series of statement syntaxes for granting and revoking these rights to users. This important area of database management is generally referred to as *SQL access privileges*.

The objects affected by SQL access privileges are tables, views and stored procedures.

- The GRANT statement assigns, to a specified user, to a role, or to another object, access privileges for a table or view or EXECUTE rights for a procedure.
- GRANT can also assign roles to users.
- The REVOKE statement removes rights.
- GRANT...WITH GRANT OPTION assigns to a user the right to grant and revoke those privileges for other users.
- GRANT ROLE...WITH ADMIN OPTION assigns users of a role the right to grant the role to others



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Privileges available

The following table lists the SQL access privileges that can be granted and revoked:

TABLE 22–3 SQL access privileges

Privilege	Access
ALL	Select, insert, update, delete data, and refer to a primary key from a foreign key
SELECT	Read data
INSERT	Write new data
UPDATE	Modify existing data
DELETE	Delete data
EXECUTE	Execute a stored procedure or call it using SELECT
REFERENCES	Refer to a primary key from a foreign key
<i>role</i>	All privileges assigned to the role. Once a role exists and has privileges assigned to it, it becomes a privilege that can be granted to users.

The ALL keyword

The ALL keyword provides a mechanism for packaging SELECT, DELETE, INSERT, UPDATE, and REFERENCES privileges in a single assignment.

❖ ALL excludes granting a role or an EXECUTE privilege.

SELECT, DELETE, INSERT, UPDATE, and REFERENCES privileges can also be granted or revoked singly or in other combinations.

❖ Statements that grant or revoke either the EXECUTE privilege or a role cannot grant or revoke other privileges.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

The PUBLIC user

The PUBLIC keyword can be used to package all users for the granting of a privilege.

❖ However, note

- PUBLIC refers only to users—stored procedures, triggers, roles and views are not included
- privileges granted to users through PUBLIC can only be revoked through PUBLIC

SQL Roles

Firebird implements features for assigning SQL privileges to groups of users, fully supporting SQL group-level security as described in the *ISO-ANSI Working Draft for Database Language SQL* sections 11.54. role definition, 11.53. GRANT statement, 11.58. REVOKE statement, and 11.57. DROP ROLE statement. It partly supports section 11.55 GRANT ROLE and 11.56 REVOKE ROLE.

- The users defined in the security database `isc4.gdb` “exist” at server level. It is thus possible to assign multiple database privileges to the same user.
- In contrast, roles are created within a database and apply only to that database.

Once a role exists and has privileges assigned to it, it becomes a privilege that can be granted to users.

Implementing roles in a database

Implementing roles is a four-step process.

- 1 Create a role using the CREATE ROLE statement.
- 2 Assign privileges to the role using GRANT *privilege* TO *rolename*.
- 3 Grant the role to users using GRANT *rolename* TO *user*.
- 4 Specify the role when attaching to a database.



These steps are described in detail in this chapter. For detailed syntax information about the [CONNECT](#), [CREATE ROLE](#), [GRANT](#), and [REVOKE](#) statements, see the *Firebird Reference Guide*—[SQL Statement and Function Reference](#) (ch. 2 p. 43).



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Granting privileges

Access privileges can be granted on an entire table or view. It is also possible to restrict UPDATE and REFERENCES privileges to specified columns.

Granting privileges for a whole table

Use GRANT to give to a user, role or stored procedure privileges for a table or view. A GRANT statement requires the following parameters, as a minimum:

- A privilege or list of privileges
- The object for which the privilege is granted
- The name of a user, role or object to which the privilege is granted

The **access privileges** can be

- a comma-separated list of one or more SELECT, INSERT, UPDATE, DELETE, REFERENCES
- the keyword ALL
- a role to which one or more privileges have been assigned.

The **object name** is the name of a table or view to which the current user has GRANT rights—as either owner, SYSDBA, operating system Superuser or as a user who has WITH GRANT OPTION privileges for the object.

The **user name**

- generally is a user which is defined in the USERS table of the Firebird security database, `isc4.gdb`.
- on all-Linux/UNIX client networks, it can also be a user which is in `/etc/password` on both the server and client machines.
- can be a stored procedure, trigger, or role.

The syntax for granting privileges to a table or view is:

```
GRANT {  
    <privileges> ON [TABLE] {tablename | viewname}  
        TO {<object> | <userlist> | GROUP UNIX_group}  
        | <role_granted> TO {PUBLIC | <role_grantee_list>} };  
  
<privileges> = ALL [PRIVILEGES] | <privilege_list>
```



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

```
<privilege_list> = {
    SELECT
    | DELETE
    | INSERT
    | UPDATE [(col [, col ...])]
    | REFERENCES [(col [, col ...])]
}
[ , <privilege_list> ...]

<object> = {
    PROCEDURE procname
    | TRIGGER trigname
    | VIEW viewname
    | PUBLIC
}
[ , <object> ...]

<userlist> = {
    [USER] username
    | rolename
    | UNIX_user
}
[ , <userlist> ...]
[WITH GRANT OPTION]

<role_granted> = rolename [ , rolename ...]

<role_grantee_list> = [USER] username [ , [USER] username ...]
[WITH ADMIN OPTION]
```

 Notice that this syntax includes the provisions for restricting UPDATE or REFERENCES to certain columns, discussed in detail in [Granting UPDATE rights for columns](#).



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Some examples

The following statement grants SELECT privilege for the DEPARTMENTS table to a user, CHALKY:

```
GRANT SELECT ON DEPARTMENTS TO CHALKY;
```

The next example grants REFERENCES privileges on DEPARTMENTS to CHALKY, permitting CHALKY to create a foreign key that references the primary key of the DEPARTMENTS table, even though he doesn't own that table:

```
GRANT REFERENCES ON DEPARTMENTS (DEPT_NO) TO CHALKY;
```

Granting UPDATE rights for columns

If you do not specify a column list, the privilege is granted for all columns in the table. GRANT can assign UPDATE or REFERENCES privileges for certain columns of a table or view. To specify the columns, place the comma-separated list of columns in parentheses following the privileges to be granted in the GRANT statement.

The following statement assigns UPDATE access to all users for the CONTACT and PHONE columns in the CUSTOMERS table:

```
GRANT UPDATE (CONTACT, PHONE) ON CUSTOMERS TO PUBLIC;
```

You can add to the rights already assigned to users at the table level, but you cannot subtract from them. To restrict user access to a table, use the REVOKE statement.

★ Views offer an elegant way to restrict access to tables, by restricting the columns and/or the rows that are visible to the user.

See [Granting privileges for views](#) for more information.

Granting privileges for a stored procedure or trigger

A stored procedure, view, or trigger sometimes needs privileges to access a table or view that has a different owner. To grant privileges to a stored procedure, put the PROCEDURE reserved word before the procedure name. Similarly, to grant privileges to a trigger or view, put the TRIGGER or VIEW reserved word before the object name.

★ When a trigger, stored procedure or view needs to access a table or view, it is sufficient for either the accessing object or the user who is executing it to have the necessary permissions.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

In the following statement the procedure COUNT_CHICKENS is granted the INSERT privilege for the PROJECTIONS table:

```
GRANT INSERT ON PROJECTIONS TO PROCEDURE COUNT_CHICKENS;
```

- ★ As a security measure, privileges to tables can be granted to a procedure instead of to individual users. If a user has EXECUTE privilege on a procedure that accesses a table, then the user does not need privileges to the table.

Multiple privileges and multiple grantees

It is possible to grant several privileges in one statement and to grant one or more privileges to multiple users or objects.

Granting multiple privileges

To give a user or object several privileges on a table, separate the granted privileges with commas in the GRANT statement. For example, the following statement assigns INSERT and UPDATE privileges for the DEPARTMENTS table to a user, CHALKY:

```
GRANT INSERT, UPDATE ON DEPARTMENTS TO CHALKY;
```

To grant a set of privileges to a procedure, place the PROCEDURE keyword before the procedure name. Similarly, to grant privileges to a trigger or view, precede the object name with the TRIGGER or VIEW keyword.

The following statement assigns INSERT and UPDATE privileges for the BUDGET table to the COUNT_CHICKENS procedure:

```
GRANT INSERT, UPDATE ON BUDGET TO PROCEDURE COUNT_CHICKENS;
```

- ★ Note that

- The GRANT statement can assign any combination of SELECT, DELETE, INSERT, UPDATE, and REFERENCES privileges.
- EXECUTE privileges must be assigned in a separate statement.
- REFERENCES privileges cannot be assigned for views.



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

Granting all privileges

The ALL privilege combines the SELECT, DELETE, INSERT, UPDATE, and REFERENCES privileges for a table into a single keyword to allow that package of privileges to be assigned to a user or procedure in one statement.

For example, the following statement grants all access privileges for the DEPARTMENTS table to a user, CHALKY:

```
GRANT ALL ON DEPARTMENTS TO CHALKY;
```

CHALKY can now perform SELECT, DELETE, INSERT, UPDATE, and REFERENCES operations on the DEPARTMENTS table.

Procedures can be assigned ALL privileges. When a procedure is assigned privileges, the PROCEDURE keyword must precede its name. For example, the following statement grants all privileges for the BUDGET table to the procedure, COUNT_CHICKENS:

```
GRANT ALL ON BUDGET TO PROCEDURE COUNT_CHICKENS;
```

Granting privileges to multiple users

There are a number of techniques available for granting privileges to multiple users. You can grant the privileges to

- a list of users
- a UNIX group
- all users (PUBLIC)
- a role, which you then assign to a user list, a UNIX group, or to PUBLIC.

Granting privileges to a list of users

To assign the same access privileges to a number of users in a single statement, provide a comma-separated list of users in place of the single user name.

For example, the following statement gives INSERT and UPDATE privileges for the DEPARTMENTS table to users MICKEY, DONALD, and GLADSTONE:

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

```
GRANT INSERT, UPDATE ON DEPARTMENTS TO MICKEY, DONALD, GLADSTONE;
```

Granting privileges to a UNIX group

Operating system *account names* on Linux/UNIX are accessible to Firebird security. A client running as a UNIX user adopts that user identity in the database, even if the account is not defined in the Firebird security database (*isc4.gdb*).

Linux/UNIX *groups* share this behavior: database administrators can assign SQL privileges to UNIX groups through SQL GRANT/REVOKE statements. This allows any OS-level account that is a member of the group to inherit the privileges that have been given to the group. For example:

```
GRANT UPDATE ON table1 TO GROUP group_name;
```

where *group_name* is a UNIX-level group defined in */etc/group*.

★ Integration of UNIX groups with database security is not a SQL standard feature.

Granting privileges to all users

To assign the same access privileges for a table to all users, use the PUBLIC keyword in preference to listing users individually in the GRANT statement.

The following statement grants SELECT, INSERT, and UPDATE privileges on the DEPARTMENTS table to all users:

```
GRANT SELECT, INSERT, UPDATE ON DEPARTMENTS TO PUBLIC;
```

❖ PUBLIC grants privileges only to users, not to stored procedures, triggers, roles, or views.

Privileges granted to users with PUBLIC can only be revoked from PUBLIC.

Granting privileges to a list of procedures

To assign privileges to several procedures in a single statement, provide a comma-separated list of procedures following the keyword PROCEDURE in the GRANT statement.

The following statement gives INSERT and UPDATE privileges for the BUDGET table to the procedures, CALCULATE_ODDS, and COUNT_BEANS:

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

```
GRANT INSERT, UPDATE ON BUDGET TO PROCEDURE CALCULATE_ODDS,  
COUNT_BEANS;
```

Granting privileges through roles

Privileges can be conferred through the use of roles. Acquiring privileges through a role is a four-step process.

- 1 Create a role using the CREATE ROLE statement.

```
CREATE ROLE rolename;
```

- 2 Assign one or more privileges to that role using GRANT.

```
GRANT privilegelist TO rolename;
```

- 3 Use the GRANT statement once again to grant the role to one or more users.

```
GRANT rolename ON table TO userlist;
```

The role can be granted WITH ADMIN OPTION, which allows users to grant the role to others, just as the WITH GRANT OPTION allows users to grant privileges to others.

- 4 At connection time, specify the role whose privileges you want to acquire for that connection.

```
CONNECT 'database' USER 'username' PASSWORD 'password'  
ROLE 'rolename';
```

Use REVOKE to remove privileges that have been granted to a role or to remove roles that have been granted to users.

❖ If you drop a role using the DROP ROLE statement, all privileges that were conferred by that role are revoked.

📖 See the *Using Firebird—SQL Statement and Function Reference* (ch. 2 p. 43) for more information on CONNECT, CREATE ROLE, DROP ROLE, GRANT, and REVOKE.

Granting privileges to a role

Once a role has been defined, you can grant privileges to that role, just as you would to a user.

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

The syntax is as follows:

```
GRANT <privileges> ON [TABLE] {tablename | viewname}
    TO rolename;

<privileges> = ALL [PRIVILEGES] | <privilege_list>

<privilege_list> = {
    SELECT
    | DELETE
    | INSERT
    | UPDATE [(col [, col ...])]
    | REFERENCES [(col [, col ...])]
}
[ , <privilege_list> ...]
```

Granting a role to users

When a role has been defined and has been granted privileges, you can grant that role to one or more users, who then acquire the privileges that have been assigned to the role.

To permit users to grant the role to others, add WITH ADMIN OPTION to the GRANT statement when you grant the role to the users.

The syntax is as follows:

```
GRANT {rolename [, rolename ...]} TO {PUBLIC
    | {[USER] username [, [USER] username ...]} } [WITH ADMIN OPTION];
```

The following example creates the MAITRE_D role, grants ALL privileges on DEPARTMENTS to this role, and grants the MAITRE_D role to HORTENSE, who then has SELECT, DELETE, INSERT, UPDATE, and REFERENCES privileges on DEPARTMENTS.

```
CREATE ROLE MAITRE_D;
GRANT ALL ON DEPARTMENTS TO MAITRE_D;
GRANT MAITRE_D TO HORTENSE;
```



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Granting users the right to grant privileges

Initially, only the owner of a table or view can grant access privileges on the object to other users. Add the WITH GRANT OPTION clause to the end of a GRANT statement to transfer the right to grant privileges to other users.

The following statement assigns SELECT access to user MINNIE and allows MINNIE to grant SELECT access to other users:

```
GRANT SELECT ON DEPARTMENTS TO MINNIE WITH GRANT OPTION;
```

❖ You cannot assign the WITH GRANT OPTION to a stored procedure.

WITH GRANT OPTION clauses are cumulative, even if issued by different users. For example, MINNIE can be given grant authority for SELECT by one user, and grant authority for INSERT by another user. Read on for more information about cumulative privileges.

Restrictions

There are only three conditions under which a user can grant access privileges (SELECT, DELETE, INSERT, UPDATE, and REFERENCES) for tables to other users or objects:

- Users can grant privileges to any table or view that they own.
- Users can grant any privileges on another owner's table or view when they have been assigned those privileges WITH GRANT OPTION.
- Users can grant privileges that they have acquired by being granted a role WITH ADMIN OPTION.

For example, in an earlier GRANT statement, MINNIE was granted SELECT access to the DEPARTMENTS table WITH GRANT OPTION. MINNIE can grant SELECT privilege to other users. Suppose MINNIE is now given INSERT access as well, but *without* the WITH GRANT OPTION:

```
GRANT INSERT ON DEPARTMENTS TO MINNIE;
```

MINNIE can SELECT from and INSERT to the DEPARTMENTS table. She can grant SELECT privileges to other users, but cannot assign INSERT privileges.

To change a user's existing privileges to include grant authority, issue a second GRANT statement that includes the WITH GRANT OPTION clause.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

For example, to allow MINNIE to grant INSERT privileges on DEPARTMENTS to others, reissue the GRANT statement and include the WITH GRANT OPTION clause:

```
GRANT INSERT ON DEPARTMENTS TO MINNIE WITH GRANT OPTION;
```

Unintended effects

Consider every extension of grant authority with care. Once other users are permitted grant authority on a table, they can grant those same privileges, as well as grant authority for them, to other users. As the number of users with privileges and grant authority for a table increases, the likelihood that different users can grant the same privileges and grant authority to any single user also expands. Quite simply, it can go out of control.

SQL assumes that any duplication of privilege or authority assignment is intentional and permits it. Duplicate privilege and authority assignments to a single user can be difficult to unravel if that user's privileges and authorities subsequently need to be revoked.



For more information about revoking privileges, see [Revoking user access](#).

For example, suppose two users to whom the appropriate privileges and grant authority have been extended, SERENA and PARKING, both issue the following statement:

```
GRANT INSERT ON DEPARTMENTS TO BRUNHILDE WITH GRANT OPTION;
```

Later, SERENA revokes the privilege and grant authority for BRUNHILDE:

```
REVOKE INSERT ON DEPARTMENTS FROM BRUNHILDE;
```

SERENA now believes that BRUNHILDE no longer has INSERT privilege and grant authority for the DEPARTMENTS table. The immediate effect of the statement is nil, because BRUNHILDE retains the INSERT privilege and grant authority assigned by PARKING.

- ❖ Grant authority should be assigned conservatively if full control of access privileges on a table is desired. In cases where privileges must be universally revoked for a user who might have received rights from several users, there are two options:
 - In order for all instances of privilege assignment to be revoked, each user who assigned rights must issue its own corresponding REVOKE statement.

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

- The table's owner must issue a REVOKE statement for all users of the table, then issue GRANT statements to re-establish access privileges for the users who need to keep their rights.



For more information about the REVOKE statement, see [Revoking user access](#).

Granting privileges to execute stored procedures

To use a stored procedure, users or other stored procedures must have EXECUTE privilege for it, using the following GRANT syntax:

```
GRANT EXECUTE ON PROCEDURE procname TO {<object> | <userlist>}
```

```
<object> = {
    PROCEDURE procname
    | TRIGGER trigname
    | VIEW viewname
    | PUBLIC
}
[, <object> ...]
```

```
<userlist> = {
    [USER] username
    | rolename
    | UNIX_user
}
[, <userlist> ...]
[WITH GRANT OPTION]
```

One stored procedure or trigger requires EXECUTE privileges on another whenever its owner is different to the owner of the procedure it wants to call.

❖ Don't overlook these cases:

- A trigger is owned by the owner of the table to which the trigger belongs
- If your statement is granting privileges to PUBLIC, it is not possible to specify additional users or objects as grantees in the same statement

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

- Users, roles or objects needing READ rights to a stored procedures that returns a data set (a "selectable stored procedure") require the EXECUTE privilege for that procedure

The following statement grants EXECUTE privilege for the CALCULATE_BEANS procedure to two users, FLATFOOT and KILROY, and to two procedures, DO_STUFF and ABANDON_OLD:

```
GRANT EXECUTE ON PROCEDURE CALCULATE_BEANS TO FLATFOOT, KILROY, PROCEDURE
DO_STUFF, ABANDON_OLD;
```

Granting privileges for views

To a user, a view looks—and often behaves—just like a table. However, the contents of a view are not stored anywhere in the database. It is the *query on the underlying base tables* that is stored. Any UPDATE, DELETE, INSERT to a view is actually a write to the table from which the view derives.

- SELECT privileges can be granted on a view just as they are on a table, since reading data from a view does not manipulate data.
- You can assign UPDATE, INSERT, and DELETE privileges to *updatable* views, just as you can to tables. UPDATES, INSERTS, and DELETES to a view are made to the view's base tables.
- REFERENCES privileges are never applicable to views.

Using views to restrict data access

Views offer a useful alternative or adjunct to SQL privileges when you need to restrict data access at both row and column levels. Because a view is usually created as a subset of columns and rows from one or more underlying tables, it provides a highly flexible way to limit the visibility and accessibility of table data.

For example, suppose an EMPLOYEES table contains the columns, LAST_NAME, FIRST_NAME, JOB, SALARY, DEPT, and PHONE. Much of this information that is useful to all employees but, in most organizations, SALARY is confidential. A view that excludes the SALARY column can provide the general information from the EMPLOYEES table, while making the SALARY data both invisible and inaccessible.

:

**Symbols** [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

```
CREATE VIEW EMPDATA AS  
SELECT LAST_NAME, FIRST_NAME, DEPARTMENT, JOB, PHONE  
FROM EMPLOYEES;
```

SELECT access to the view, EMPDATA, can be granted to everyone without compromising confidentiality or permitting unauthorized access to the protected SALARY column.

- ❖ Care is still necessary in designing a view involving base tables that contain sensitive information.
Don't make it possible for users to recreate or infer the missing data from those published in the view.

For more information about working with views, see chapter 19, [Views](#) (p. 363)

Updatable vs non-updatable views

Views deriving from a single table, without aggregates or reflexive (self) joins, are often updatable, since it is possible to map the output rows and columns uniquely to those from which they derive. Any view derived from a join or an aggregate is considered to be a *read-only* or *non-updatable* view, since column-for-column and row-for-row mappings cannot be made directly.

For more information about this topic, see [Read-only and updatable views](#) on p. 368 of chapter 19.

It is meaningful to grant INSERT, UPDATE, and DELETE privileges for a view *only* if the view is updatable.

- ❖ Although you can grant these privileges to a read-only view without receiving an error message, any actual write operation fails because the view is read-only.
- ★ You can use triggers to simulate updating a read-only view. Be aware, however, that any triggers you write are subject to all the integrity constraints on the base tables. To see an example of how to use triggers to "update" a read-only view, see [Making read-only views updatable](#) on p. 370 of chapter 19.



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

WITH CHECK OPTION

When creating a view for which you plan to grant INSERT and UPDATE privileges, include the WITH CHECK OPTION constraint to restrict users to updating only those rows from the underlying table which are accessible through the view.

- ❖ Views created using the WITH CHECK OPTION integrity constraint can be updated only if the UPDATE statement fulfills the constraint's requirements.

Effects of DML operations on updatable views

When a view is based on a single table, data changes are made directly to the view's underlying base table.

- UPDATE affects only the base table columns selected through the view. Values in other columns are invisible to the view and its users and are never changed.
- DELETE removes a row from the view, thereby removing the entire row from the base table, including columns not visible to the view.
 - ❖ A DELETE statement will fail if the deletion of the row violates any integrity constraints or trigger conditions in the underlying table.
- An INSERT operation adds a row to the base table with all columns, including those not visible to the view. Inserting a row into a view succeeds only when:
 - Data being inserted into the columns visible to the view meet all existing integrity constraints and trigger conditions for those columns.
 - All other columns of the base table are allowed to contain NULL values.

Syntax

The syntax for granting privileges to a view is:

```
GRANT <privileges> ON viewname  
TO {<object> | <userlist> | GROUP UNIX_group};
```



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

<privileges> = ALL [PRIVILEGES] | *<privilege_list>*

<privilege_list> = {

SELECT

| DELETE

| INSERT

| UPDATE [(*col* [, *col* ...])]

}

[, *<privilege_list>* ...]

<object> = {

PROCEDURE *procname*

| TRIGGER *trigname*

| VIEW *viewname*

| PUBLIC

}

[, *<object>* ...]

<userlist> = {

[USER] *username*

| *rolename*

| UNIX_user

}

[, *<userlist>* ...]

[WITH GRANT OPTION]

Revoking user access

Use the REVOKE statement to remove privileges that were assigned with the GRANT statement. A REVOKE requires, as a minimum, the following parameters:

- One access privilege to remove
- The table or view to which the privilege revocation applies

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

- The name of the grantee for which the privilege is revoked.

In its full form, REVOKE removes all the privileges that GRANT can assign.

Syntax

```
REVOKE <privileges> ON [TABLE] {tablename | viewname}
    FROM {<object> | <userlist> | GROUP UNIX_group};

<privileges> = ALL [PRIVILEGES] | <privilege_list>

<privilege_list> = {
    SELECT
    | DELETE
    | INSERT
    | UPDATE [(col [, col ...])]
    | REFERENCES [(col [, col ...])]
}

[, <privilege_list> ...]

<object> = {
    PROCEDURE procname
    | TRIGGER trigname
    | VIEW viewname
    | PUBLIC
}
[, <object> ...]

<userlist> = [USER] username [, [USER] username ...]
```

The following statement removes the SELECT privilege for the user, KILROY, on the DEPARTMENTS table:

```
REVOKE SELECT ON DEPARTMENTS FROM KILROY;
```

The following statement removes the UPDATE privilege for the procedure, COUNT_BEANS, on the ACCOUNTS table:

```
REVOKE UPDATE ON ACCOUNTS FROM PROCEDURE COUNT_BEANS;
```

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

The next statement removes EXECUTE privilege for the procedure, COUNT_BEANS, on the ABANDON_OLD procedure:

```
REVOKE EXECUTE ON PROCEDURE ABANDON_OLD FROM PROCEDURE COUNT_BEANS;
```

For the complete syntax of REVOKE, see the *Firebird Reference Manual*.

Restrictions when revoking

The following restrictions and rules of scope apply to the REVOKE statement:

- Privileges can be revoked only by the user who granted them.
- Other privileges assigned by other users are not affected.
- Revoking a privilege for a user, A, to whom grant authority was given, automatically revokes that privilege for all users to whom it was subsequently assigned by user A.
- Privileges granted to PUBLIC can only be revoked for PUBLIC.

Revoking multiple privileges

To remove some, but not all, of the access privileges assigned to a user or procedure, list the privileges to remove, separating them with commas. For example, the following statement removes the INSERT and UPDATE privileges for the DEPARTMENTS table from a user, SERENA:

```
REVOKE INSERT, UPDATE ON DEPARTMENTS FROM SERENA;
```

The next statement removes INSERT and DELETE privileges for the ACCOUNTS table from a stored procedure, COUNT_BEANS:

```
REVOKE INSERT, DELETE ON ACCOUNTS FROM PROCEDURE COUNT_BEANS;
```

Any combination of previously assigned SELECT, DELETE, INSERT, and UPDATE privileges can be revoked.

Revoking all privileges

The ALL privilege combines the SELECT, DELETE, INSERT, and UPDATE privileges for a table in a single expression. It is a shorthand way to remove all SQL table access privileges from a user or procedure. For example, the following statement revokes all access privileges for the DEPARTMENTS table for a user, MAGPIE:

```
REVOKE ALL ON DEPARTMENTS FROM MAGPIE;
```

❖ Observe that

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

- ALL does not revoke EXECUTE privilege.
- If you drop a role using the DROP ROLE statement, all privileges that were conferred by that role are revoked.

Users with unknown privileges

ALL can be used even if a user does not have all access privileges for a table . Using ALL in this manner is helpful when a current user's access rights are unknown.

Revoking privileges for a list of users

Use a comma-separated list of users to REVOKE access privileges for a number of users in a single statement. The following statement revokes INSERT and UPDATE privileges on the DEPARTMENTS table for users MAGPIE, BRUNHILDE, and KILROY:

```
REVOKE INSERT, UPDATE ON DEPARTMENTS FROM MAGPIE, BRUNHILDE, KILROY;
```

Revoking privileges for a role

If you have granted privileges to a role or granted a role to users, you can use REVOKE to remove the privileges or the role.

To remove privileges from a role

```
REVOKE privileges ON table FROM rolenamelist;
```

To revoke a role from users

```
REVOKE role_granted FROM {PUBLIC | role_grantee_list};
```

The following statement revokes UPDATE privileges from the CARTEBLANCHE role:

```
REVOKE UPDATE ON DEPARTMENTS FROM CARTEBLANCHE;
```

Now, users who were granted the CARTEBLANCHE role no longer have UPDATE privileges on DEPARTMENTS, although they retain the other privileges—SELECT, INSERT, DELETE, and REFERENCES—that they acquired with this role.

Revoking a role from users

Use REVOKE to remove a role that you assigned to users.

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

The following statement revokes the CARTEBLANCHE role from KILROY:

```
REVOKE CARTEBLANCHE FROM KILROY;
```

KILROY no longer has any of the access privileges that she acquired as a result of membership in the CARTEBLANCHE role. However, if any others users have granted the same privileges to the user KILROY, he still retains them.

Revoking EXECUTE privileges

Use REVOKE to remove EXECUTE privileges on a stored procedure. The syntax for revoking EXECUTE privileges is as follows:

```
REVOKE EXECUTE ON PROCEDURE procname FROM {<object> | <userlist>}  
<object> = {  
    PROCEDURE procname  
    | TRIGGER trigname  
    | VIEW viewname  
    | PUBLIC  
}  
[ , <object> ...]  
<userlist> = [USER] username [ , [USER] username ...]
```

The following statement removes EXECUTE privilege for user MINNIE on the COUNT_CHICKENS procedure:

```
REVOKE EXECUTE ON PROCEDURE COUNT_CHICKENS FROM MINNIE;
```

Revoking privileges from objects

REVOKE can remove the access privileges for one or more procedures, triggers, or views. Precede each type of object by the appropriate keyword (PROCEDURE, TRIGGER, or VIEW). Make a separate comma-separated list for each object type. The object type keyword will mark the start of each these list.

The following statement revokes INSERT and UPDATE privileges for the ACCOUNTS table from the COUNT_CHICKENS and ABANDON_OLD procedures and from the SHOW_USER trigger.

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

```
REVOKE INSERT, UPDATE ON ACCOUNTS FROM PROCEDURE COUNT_CHICKENS,  
aABANDON_OLD TRIGGER SHOW_USER;
```

Revoking privileges from user PUBLIC

To revoke privileges granted to all users as PUBLIC, use REVOKE with PUBLIC. For example, the following statement revokes SELECT, INSERT, and UPDATE privileges on the DEPARTMENTS table for all users:

```
REVOKE SELECT, INSERT, UPDATE ON DEPARTMENTS FROM PUBLIC;
```

When this statement is executed, only the table's owner retains full access privileges to DEPARTMENTS.

❖ Observe that revoking privileges from PUBLIC

- does not revoke privileges for stored procedures.
- can not be used to strip privileges from users who were granted them as individual users.

Revoking grant authority

To revoke a user's grant authority for a given privilege, use the following REVOKE syntax:

```
REVOKE GRANT OPTION FOR privilege [, privilege ...] ON table  
FROM user;
```

For example, the following statement revokes SELECT grant authority on the DEPARTMENTS table from a user, MINNIE:

```
REVOKE GRANT OPTION FOR SELECT ON DEPARTMENTS FROM MINNIE
```

[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

CHAPTER 23

Server and Database Statistics

The Firebird server keeps a number of statistics that it uses in the management and decision control processes for running databases. Because some of these statistics are of interest to the humans who administer databases, Firebird provides tools for retrieving reports from the internal data structures in which it maintains these statistics. In this chapter, the utility `gstat` and the lock report utilities are considered.

- `gstat` reports on the performance characteristics of tables and/or indexes. This information can provide useful guidance as to where performance bottlenecks are occurring because of sub-optimal metadata structures.
- The *lock print utilities*, `iblockpr` (Windows) and `gds_lock_pr` (Linux/UNIX) retrieve reports from the internal lock table that Firebird maintains to control transaction concurrency. This information is useful when you are designing and testing multi-user scenarios to trap, handle or avoid deadlock conditions.

gstat command-line tool

`gstat` is a command-line tool which can be used by SYSDBA or the owner of a database to retrieve statistics reports from a database.

You need to run `gstat` on the server host, since it works only database files that are located on the same machine from which you launch the program.

gstat on Linux/UNIX

Because `gstat` accesses database files at the filesystem level, rather than through the Firebird server, it is necessary on Linux/UNIX platforms to have system-level read access to the file. You can achieve this in one of two ways, either:

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

- log in to the account which is running the server (root, firebird or interbas[e])
- set the system-level permissions on the database file to include read permission for your Group.

The gstat interface

Unlike some of the other command-line tools, gstat does not have its own shell interface. Each request involves calling gstat with switches.

Syntax

```
gstat [switches] db_name
```

db_name is the fully qualified local path to the database you want to query.

Possible switches

TABLE 23-1 gstat switches

Switch	Description
<code>-user <i>username</i></code>	Checks for user <i>username</i> before accessing database
<code>-pa[ssword] <i>password</i></code>	Checks for password <i>password</i> before accessing database
<code>-header</code>	Report the information on the header page, then stop reporting
<code>-log</code>	Report the information on the header and log pages, then stop reporting
<code>-index</code>	Retrieve and display statistics on indexes in the database
<code>-data</code>	Retrieve and display statistics on user data tables in the database
<code>-all</code>	This is the default report if you do not request -index, -data or -all. It retrieves and displays statistics from both -index and -data



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

TABLE 23-1 gstat switches

Switch	Description
-system	As -all (above) but additionally includes statistics on the system tables
-r	Retrieves and displays record size and version statistics (including back versions)
-t <i>table_list</i>	Used in conjunction with -data, restricts output to the tables listed in <i>table_list</i>
-z	Print product version of gstat

gstat switches in detail

The -data switch

```
gstat -data db_name
```

Retrieves a table-by-table database summary displaying information about data pages. To include the system tables (RDB\$XXX) in the report, add the **-system** switch.

Example of data page summary output

```
> gstat -data c:\databases\MyDB.gdb
```

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z****COUNTRY (31)**

Primary pointer page: 246, Index root page: 247

Data pages: 1, data page slots: 1, average fill: 59%

Fill distribution:

0 - 19% = 0

20 - 39% = 0

40 - 59% = 1

60 - 79% = 0

80 - 99% = 0

The first line displays a database table name while the remaining lines contain item information pertaining to that table:

TABLE 23–2 gstat —Data output

Item	Description
Primary pointer page	The page that is the first pointer page for the table.
Index root page	The page number that is the first pointer page for indexes.
Data pages	The total number of data pages.
Data page slots	The number of pointers to database pages, regardless of whether the pages are still in the database
Average fill	The average percentage to which the data pages are filled.
Fill distribution	A histogram showing the number of data pages that are filled for each percentage block, by blocks of 20%.

Restricting the output from gstat –data

If you don't want a **gstat** of every table, the **-t** switch allows you to specify a list of tables in which you are interested.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- ❖ The Firebird 1.0 version of **gstat** does not support the **-t** switch on databases that use quoted identifiers and case-sensitivity for table identifiers.

Syntax for the **-t table_list** switch is

```
gstat -data db_name -t tablename [tablename2 tablename3 ...]
```

Table names must be typed *in uppercase*.

The **-header** switch

```
gstat -header db_name
```

Displays a database summary showing the header page information.

Example of header page summary output

```
> gstat -header c:\databases\MyDB.gdb
Database 'C:\databases\MyDB.gdb'
Database header page information:
Flags 0
Checksum 15351
Generation 174
Page size 1024
ODS version 9.0
Oldest transaction 22
Oldest active 166
Oldest snapshot 166
Next transaction 170
Bumped transaction 1
Sequence number 0
Next attachment ID 0
Implementation ID 2
...
```

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

Shadow count 0
Page buffers 0
Next header page 0
Database dialect 3
Creation date Nov 10, 1999 16:24:31
Attributes force write
Variable header data:

The first line displays the name and location of the primary database file. The following lines contain information from the database header page:

TABLE 23-3 gstat—Header page output

Item	Description
Flags	..
Checksum	The header page checksum. In ancestor versions of InterBase, it was a unique value computed from all the data in the header page. It was turned off and, in Firebird, it is always 12345. When the header page is stored to disk and later read, the checksum of the retrieved page is compared to 12345 and, if it does not match, a Checksum error is raised. This catches some kinds of physical corruption.
Generation	Counter incremented each time header page is written.
Page size	The current database page size, in bytes.
ODS version	The version of the database's on-disk structure.
Oldest transaction	The transaction ID number of the oldest "interesting" transaction (those that are not committed, but active, in limbo, or rolled back).
Oldest active	The transaction ID number of the oldest active transaction.

**Symbols** [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

TABLE 23-3 gstat—Header page output

Item	Description
Oldest snapshot	..
Next transaction	The transaction ID number that Firebird assigns to the next transaction. The difference between the oldest transaction and the next transaction determines when database sweeping occurs. The default value is 20,000 . See the section on <i>Sweeping databases</i> in the chapter <i>The gfix command-line tool</i> .
Bumped transaction	Now obsolete.
Sequence number	The sequence number of the header page, starting at zero for the first page.
Next connection ID	ID number of the next database connection.
Implementation ID	The architecture of the system on which the database was created. These ID definitions are platform-dependent #define directives for a macro named CLASS and can be any of the following: <ul style="list-style-type: none">• IBM AIX POWER series, IBM AIX PowerPC• Sun Solaris SPARC, HP9000 s300, Xenix, Motorola IMP UNIX, UnixWare, NCR UNIX, NeXT, Data General DG-UX Intel• Sun Solaris x86• VMS• Ultrix VAX• Ultrix MIPS• HP9000 s700/s800• Novell NetWare



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

TABLE 23-3 gstat—Header page output

Item	Description
	<ul style="list-style-type: none">• Apple Macintosh 680x0• HP Apollo DomainOS• Data General DG-UX 88K• MPE/xl• SGI IRIX, 14 Cray• SF/1
	<ul style="list-style-type: none">• Microsoft Windows 32-bit Intel• IBM OS/2• Microsoft Windows 16-bit• Linux Intel• Linux SPARC
Shadow count	The number of shadow files defined for the database.
Number of cache buffers	The number of page buffers in the database cache.
Next header page	The ID of the next header page.
.Database dialect	The SQL dialect of the database
Creation date	The date when the database was created.
Attributes	<ul style="list-style-type: none">• force write—indicates that forced database writes are enabled.• no_reserve—indicates that space is not reserved on each page for old generations of data.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

TABLE 23-3 gstat—Header page output

Item	Description
Attributes (cont.)	This enables data to be packed more closely on each page and therefore makes the database occupy less disk space. <ul style="list-style-type: none">• shutdown—indicates database is shut down.
Variable header data	<ul style="list-style-type: none">• sweep interval• secondary file information

The **-index** switch

```
gstat -index db_name
```

Retrieves and displays statistics on indexes in the database:
average key length (bytes), total duplicates, and maximum duplicates for a single key.

Add the **-system** switch to include the system indexes in the report.

Example of index page summary output

```
Index CUSTNAMEEX (2)
Depth: 2, leaf buckets: 2, nodes: 27
Average data length: 45.00, total dup: 0, max dup: 0
Fill distribution:
0 - 19% = 0
20 - 39% = 0
40 - 59% = 1
60 - 79% = 0
80 - 99% = 1
```

**Symbols** [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)TABLE 23-4 **gstat**—Index page output

Item	Description
Index	The name of the index.
Depth	The number of levels in the index page tree. If the depth of the index page tree is greater than three, then sorting may not be as efficient as possible. To reduce the depth of the index page tree, increase the page size. If increasing the page size does not reduce the depth, then return it to its previous size.
Leaf buckets	The number of leaf (bottom level) pages in the index page tree.
Nodes	The total number of records indexed by in the tree.
Average data length	The average length of each key, in bytes.
Total dup	The total number of rows that share duplicate indexes.
Max dup	The number of duplicates of the index with the most duplicates
Fill distribution	A histogram showing the number of index pages that are filled for each percentage block, by blocks of 20%.

Because **gstat** performs its analysis at file level, it has no concept of transactions. Consequently, index statistics include information about indexes involved in non-committed transactions.

The **-r** switch

```
gstat -r db_name
```

Retrieves and displays record size and version statistics:

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

- For records: average record length (bytes) and total records in the table.
- For (back) versions: average version length (bytes), total versions in the table, and maximum version chain for a record.

Because **gstat** performs its analysis at file level, it has no concept of transactions. Consequently, the total number of records in a table could include both active and dead transactions

Record and version length are user data, and do not take into account the record header that prefixes every record version.

Examples of record version output

The three tables CAULDRON, CAULDRON1 and CAULDRON2 have identical metadata and cardinality of 100,000 records. The nominal, uncompressed record length is ~900 bytes.

- CAULDRON is a clean table of records with no back versions. The average stored record length is ~121 bytes: about an 87% compression efficiency.
- CAULDRON1 has an active transaction that just executed:
`DELETE FROM CAULDRON1`

Every record has a zero (0.00) length because the primary record is a *delete stub* that contains only a record header. The committed records were all restored as back versions and compressed to the same size they had when they were the primary records. The table still has the same number of pages (4,000) as before the DELETE. The average fill factor rose from 85% to 95% to house all the delete stubs.

- CAULDRON2 has an active transaction that just executed:

```
UPDATE CAULDRON2 SET F2OFFSET = 5.0
```

The updated records have each grown by two bytes (121 to 123), attributable to lower compression.

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

The value 5.0 replaced many missing and zero values, which made the field value different from its neighboring fields. There are now 100,000 back versions averaging 10 bytes each. The average fill factor has risen to 99% and the table grew 138 pages to 4,138.

```
> gstat proj.gdb -r -t CAULDRON CAULDRON1 CAULDRON2
...
Analyzing database pages ...
CAULDRON (147)
Primary pointer page: 259, Index root page: 260
Average record length: 120.97, total records: 100000
Average version length: 0.00, total versions: 0, max versions: 0
Data pages: 4000, data page slots: 4000, average fill: 85%
Fill distribution:
0 - 19% = 0
20 - 39% = 0
40 - 59% = 0
60 - 79% = 0
80 - 99% = 4000
CAULDRON1 (148)
Primary pointer page: 265, Index root page: 266
Average record length: 0.00, total records: 100000
Average version length: 120.97, total versions: 100000,
max versions: 1
Data pages: 4000, data page slots: 4000, average fill: 95%
Fill distribution:
0 - 19% = 0
20 - 39% = 0
40 - 59% = 0
60 - 79% = 0
80 - 99% = 4000
...
...
```

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z****CAULDRON2 (149)**

Primary pointer page: 271, Index root page: 272

Average record length: 122.97, total records: 100000

Average version length: 10.00, total versions: 100000, max
versions: 1

Data pages: 4138, data page slots: 4138, average fill: 99%

Fill distribution:

0 - 19% = 0

20 - 39% = 0

40 - 59% = 0

60 - 79% = 0

80 - 99% = 4138

Retrieving statistics into your programs

The Firebird API function *isc_database_info()* provides items which enable you to retrieve performance timings and database operation statistics. The following request buffer items are available:

TABLE 23-5 Database I/O statistics *isc_info* items

Request Buffer Item	Result Buffer Contents
<i>isc_info_fetches</i>	Number of reads from the memory buffer cache; calculated since the Firebird server started
<i>isc_info_marks</i>	Number of writes to the memory buffer cache; calculated since the Firebird server started
<i>isc_info_reads</i>	Number of page reads; calculated since the Firebird server started
<i>isc_info_writes</i>	Number of page writes; calculated since the Firebird server started
<i>isc_info_backout_count</i>	Number of removals of record versions per table since the current database attachment started

**Symbols** [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)TABLE 23-5 Database I/O statistics *isc_info* items

Request Buffer Item	Result Buffer Contents
isc_info_delete_count	Number of row deletions <ul style="list-style-type: none">• Reported per table• Calculated since the current database attachment started
isc_info_expunge_count	Number of removals of a record and all of its ancestors, for records whose deletions have been committed <ul style="list-style-type: none">• Reported per table• Calculated since the current database attachment started
isc_info_insert_count	Number of inserts into the database whose deletions have been committed <ul style="list-style-type: none">• Reported per table• Calculated since the current database attachment started
isc_info_purge_count	Number of removals of old versions of fully mature records (records committed, resulting in older ancestor versions no longer being needed) whose deletions have been committed <ul style="list-style-type: none">• Reported per table• Calculated since the current database attachment started
isc_info_read_idx_count	Number of reads done via an index whose deletions have been committed <ul style="list-style-type: none">• Reported per table• Calculated since the current database attachment started



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

TABLE 23-5 Database I/O statistics *isc_info* items

Request Buffer Item	Result Buffer Contents
<code>isc_info_read_seq_count</code>	Number of sequential database reads, that is, the number of sequential table scans (row reads) whose deletions have been committed <ul style="list-style-type: none">• Reported per table• Calculated since the current database attachment started
<code>isc_info_read_update_count</code>	Number of row updates whose deletions have been committed <ul style="list-style-type: none">• Reported per table• Calculated since the current database attachment started



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

Lock statistics

Locking in Firebird

Firebird uses a locking mechanism to maintain the consistency of the database when it is accessed by multiple users. The lock manager is a thread in the `ibserver` process that coordinates locking. It maintains a *lock table* to coordinate resource sharing among the client threads in the process that are connected to the database. In this table it stores information which can be useful when trying to correct deadlock situations, for example:

- information on all the locks in the system, including their states
- global header information aggregating statistics such as the size of the lock table, the number of free locks, the number of deadlocks, and so on.
- process flags, indicating such information as whether the lock has been granted or is waiting

The Lock Print utilities

The programs that extract the lock table statistics are named `iblockpr` on Windows and `gds_lock_pr` on Linux/UNIX. Each does the same work on its respective platform.

Syntax 1

The first form of syntax retrieves a static report of lock statistics at one instant in time:

`iblockpr [a,o,w]` (Windows) or `gds_lock_print [a,o,w]` (Linux/UNIX)

Switches apply to either `iblockpr` or `gds_lock_print`

Switches for syntax 1

With **syntax 1**, you can request these static reports:

- `-a` prints a static view of the contents of the lock table
- `-o` prints a static lock table summary and a list of all entities that own blocks. This output is the same as if you used the `-i` switch on its own.
- `-w` prints out all the information provided by the `-o` flag and adds WAIT statistics for each owner. With this report you can work out which owner's request is blocking others' in the lock table.



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

Syntax 2

The second form collects a specified number of samples at fixed intervals and produces an interactive report monitoring current lock table activity.

Sampling occurs *n* times at intervals of *t* seconds. One line is printed for each sample. The average of the sample values is printed at the end of each column.

- *t* specifies the time in seconds between samplings
- *n* specifies the number of samples to be taken.

:

```
iblockpr [-i{a,o,w}] [t n]
```

If you do not supply values for *n* and *t*, the default is *n*=1.

Switches for syntax 2

The **-i** switch on its own prints all statistics. To produce output in smaller chunks that are easier to read, you can modify this switch as follows:

- **-ia** prints the number of threads trying to acquire access to the lock table per second
- **-io** prints operating statistics such lock requests, conversions, downgrades, and releases per second
- **-iw** prints number of lock acquisitions and requests waiting per second, wait percent, and retries

Example

The following statement prints “acquire” statistics (access to lock table: acquire/s, acqwait/s, %acqwait, acqrtry/s, and rtrysuc/s) every 3 seconds until 10 samples have been taken:

```
gds_lock_print -ia 3 10
```



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

CHAPTER 24

Housekeeping & Repair: gfix

As SYSDBA or the database owner you can use the command-line administrative tool gfix to attach to a database and perform a variety of housekeeping and recovery tasks. Use gfix when you need to

- initiate a database shutdown
- switch between synchronous (forced) and asynchronous writes
- change a read-write database to read-only and vice-versa
- change the dialect
- set the size of the database cache
- display, commit or recover limbo transactions
- mend corrupted databases and data in certain conditions
- change the sweep interval
- perform a sweep
- activate and drop shadow databases

Using gfix

To run gfix, open a console window running your operating system shell and enter a command. Notice that, at present, gfix does not have its own shell interface. You need to enter each gfix command call along with the switches for the options you want to use and the name of the database you want to command to operate on.

Syntax for commands

```
gfix [switches] db_name
```

db_name must be the full name of the *primary* file of the database on which you want the operations to be performed.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- ★ The primary file of a single-file database is the database file itself. For multiple-file databases, the primary file is *first* file in the set.

Switches for gfix options

The available options for **gfix** are described in the following sections and summarized in the Firebird Reference Manual. Switches identify the operations and options.

- ★ Abbreviations can be used for most option switches. Optional characters are shown in square brackets ([]). You may include any number of the optional characters up to and including the full identifier of the switch, as long as no characters are missing in left-to-write sequence.

Getting database access with gfix

If you are connecting to the server remotely, the user name and password of either the SYSDBA user or the database owner must be included among the switches you supply. The switches are

-pas [sword] password
-user name

- ❖ Notice that the abbreviated form of the -password switch for **gfix** is different to that used in the switches for other command-line utilities.

Example of a command using password and user name

```
gfix -w sync customer.gdb -pas heureuse -user SYSDBA
```



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

User and password on a local connection

Before starting up the server in order to do extended work locally on a copy of a damaged database or on transaction recovery, you can add two operating system variables to avoid having to type the SYSDBA or owner user name and password in every command. They are:

```
shell prompt> SET ISC_USER=SYSDBA  
shell prompt> SET ISC_PASSWORD=heureuse
```

For security reasons, you should remove these environment variables as soon as you finish your task.

It is also possible to configure these as permanent environment variables. It is not recommended in a production environment, of course, because of the security risk.

Shutting down a database

Whenever you need to shut down the server in a production environment, you will normally want to shut down individual databases in a controlled way first. The means to do this is the **-shut** switch.

The **-shut** switch must be qualified by either **-attach**, **-force**, or **-tran**

Syntax for shutting down

```
gfix -sh { -at n | -f n | -t n } db_name
```

Mandatory qualifying switches for shut

The mandatory qualifying switches for use with **-shut** are:

- **-at[tach] n** Use to prevent new database connections during a time-out period of **n** seconds.
Causes shutdown to be cancelled if there are still processes connected after **n** seconds.
- **-tr[an] n** Use to prevent new transactions from starting during time-out period of **n** seconds.
Causes shutdown to be cancelled if there are still active transactions after **n** seconds.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- **-f[orce] n** Use to force shutdown of a database after *n* seconds regardless of any connected processes or active transactions.
❖ This is a drastic operation that should be used with caution.

Terminating a shutdown

The **-o[nline]** switch cancels a **-shut** operation that is scheduled to take effect, or rescinds a shutdown that is currently in effect, and puts the database back on line for multi-user access.

Syntax

```
gfix -sh -o db_name
```

Changing database settings

A number of **gfix** command options permit certain configuration settings to be set or changed for a database.

- ❖ To perform these commands, SYSDBA or owner privileges are required and **gfix** must obtain exclusive access.

Setting default cache size

The database cache is a set of memory buffers allocated for caching pages (data, index, BLOB, and so on) that client connection processes read from the database and may subsequently write back. The amount of memory reserved for the cache depends on the number of buffers (database pages) set, on page size and also on whether the Firebird server installed has the Classic or the Superserver architecture.

- Classic creates a separate server process and database cache for each client connection. The installation default for Classic is 75 pages per client.
- Superserver runs a single process for all connections, creating a separate thread for each. The database cache is pooled for all connections. The installation default for Superserver is 1024 pages.



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

Effect of varying the cache size

Increasing the cache can reduce the number of I/O operations the client may need to do in the course of a session. There is a cost in increased memory requirement. For example, 50 Classic server processes each reserving 200 2048-byte pages will consume 20 Mb of memory. Any connection can increase or reduce the number of cache buffers available.

Remember that, if you increase the page size, the cache size will rise accordingly. You should ensure that you take the amount of physical RAM on the machine into consideration when altering cache size. Once the cache reaches the point where it is too large to be kept in RAM, it will begin swapping out to disk, thereby thoroughly defeating the benefit of having a cache at all.

The database cache size is configurable by several means, of which **gfix** is just one.

- in the ibconfig file, the default can be set by the parameter `DATABASE_CACHE_SIZE`
 For more details, see [Configuring the database cache](#) on page 67 in chapter 5.
- through the API function `isc_attach_database()`, the default cache can be overridden by a specific connection specifying a value for the cache size member of the database parameter block (DPB). In Classic, this will override any default size that is currently applicable for the database, for any connection. In Superserver it will increase or decrease the cache size of the pooled cache.
 For more information, refer to the *API Guide* volume of the InterBase® 6 documentation set (*APIGuide.PDF*)
- the `gfix -b[uffers] n` option (see below) will override any default applicable through the ibconfig setting or through an earlier gfix -b command.

gfix syntax for setting the cache

`gfix -b n db_name`

where `n` is the number of database pages to be reserved.



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

Other cache options

`-ca[che] n` is currently unused, reserved for future implementation

Changing the access mode

Use the gfix -mo[de] option to switch the connection access mode between read-only and read-write.

Syntax for setting access mode

To switch from read-write to read-only:

```
gfix -mo[de] read_only db_name
```

To switch from read-only to read-write:

```
gfix -mo[de] read_write db_name
```

Example

To change the `customer.gdb` database to read-only mode:

```
gfix -mo read_only customer.gdb
```

Changing the database dialect

In the Firebird context, "dialect" refers to a set of language features which the database engine can support. Currently, Firebird can operate in three dialects:

Dialect 1 inherits the Borland InterBase 5.6 language set. An InterBase 5.x database restored using the Firebird version of `gbak` will behave in the Firebird server and your 5.x applications just as it did before.



Symbols **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

Dialect 2 is an intermediate stage for admin use during the process of converting a Dialect 1 database to Dialect 3. Using a Dialect 2 database with your application code is not recommended. You can only attach to a Dialect 2 database with **gfix**, or when the Dialect 2 attribute of the client program is set true. This is possible using the command-line **isql** tool, the Firebird API or an object interface that supports setting the dialect.

❖ Only a Dialect 1 database can be set to Dialect 2 and, once a database is in Dialect 2 or 3, it cannot be reverted to Dialect 1.

Dialect 3 supports a language set which has a number of new SQL-92 features and significant changes in several data types.

For more information, see [Migrating to Firebird](#) on page 617 in chapter 27.

Syntax for changing the dialect

The command switch is **-s[ql_dialect] n**.

```
gfix -s n db_name
```

where **n** is either 1, 2 or 3.

Example

To change the dialect of the database **customer.gdb** database to 3:

```
gfix -s 3 customer.gdb  
gfix -sql 3 customer.gdb
```

is also possible, as is

```
gfix -sql_dialect 3 customer.gdb
```

Enabling and disabling ‘Use All Space’

By default, Firebird fills database pages so that the ratio of data stored per page does not exceed 80 per cent. A certain amount of compaction can be achieved by switching the ratio to 100 percent. This may produce a

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

performance benefit during huge bulk inserts, especially if row size is smaller than page size and row size multiples can be stored economically within a one-page space.

To enable 'use all space' use the command

```
gfix -use full customer.gdb
```

To disable 'use all space' and return to the 80 per cent fill ratio, use the command

```
gfix -use reserve customer.gdb
```

★ The 'Use all space' feature can be set as a switch when restoring a database using **gbak**.

Enabling and disabling Forced Writes

Forced Writes is synonymous with *synchronous writes*. The term "disabling Forced Writes" means switching the write behavior from synchronous to asynchronous.

When the behaviour is synchronous ("Forced Writes enabled"), new records, new record versions and deletions are physically written to disk immediately upon COMMIT. Asynchronous writes ("Forced Writes disabled") hold new and changed data in the file cache, relying on the flushing behavior of the operating system to make them permanent on disk.

Firebird is installed on Windows NT/2K/XP and Linux with Forced Writes enabled. In a very robust environment with highly reliable UPS support, a DBA may disable Forced Writes to reduce I/O and improve performance. When Forced Writes is disabled in less dependable environments, the database becomes susceptible to data loss and even corruption in the event of an uncontrolled shutdown.

Forced writes are not applicable to Windows 95/98 nor ME.

SystemRestore on Windows ME and XP

Windows ME, along with the XP Home and Professional editions, has a feature named *System Restore*, which causes the operating system to update its own filesystem backup of files with certain suffixes each time a file I/O operation occurs. The suffix ".gdb" is included in *filelst.xml*, the list of files so affected, which lives in the Windows/System folder. Unfortunately, file types cannot be removed.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

❖ System Restore is *not* a substitute for forced writes.

Disabling Forced Writes on Windows servers

Currently, Windows is less dependable than other operating systems with regard to cache flushing. If Forced Writes is disabled on a 24/7 Windows server, flushing may never occur.

Working with a converted InterBase 6 database

Be aware that, if you created your Firebird database in InterBase 6.x (commercial or Open Edition) or a beta version of Firebird and attention has not been given to this setting, Forced Writes will be *disabled by default*.

Syntax for enabling and disabling Forced Writes

The command switch is `-w[rite] {sync | async}`

To enable Forced Writes:

```
gfix -w sync customer.gdb
```

To disable:

```
gfix -w async customer.gdb
```

Finding out what version of Firebird server is installed

The switch `-z` (with no parameters) shows the version of gfix and the Firebird engine installed on the server.

Syntax

```
gfix -z
```

Transaction recovery

gfix provides tools for recovering transactions left in limbo after a connection is lost during a multi-database transaction.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Two-phase commit

A transaction which spans multiple Firebird databases is committed in two steps, or *phases*. This two-phase commit guarantees that, if the transaction cannot complete the updates to all of the databases involved, it will not update any of them.

In the first phase of a two-phase commit, Firebird prepares a *sub-transaction* for each database involved in the transaction, and writes the appropriate changes to each database.

In the second phase, following the same order in which it prepared and wrote them, Firebird marks each sub-transaction as committed.

Limbo transactions

Limbo transactions are sub-transactions that remain unresolved if something traumatic happens to one or more database connections during the second phase of the two-phase commit: for example, a network fault or a power failure. The server cannot tell whether limbo transactions should be committed or rolled back. Consequently, some records in a database may become inaccessible until explicit action is taken to resolve the limbo transactions with which they are associated.

Transaction recovery

With **gfix**, you have a number of options for inquiring about and resolving limbo transactions after the traumatic failure of a two-phase commit. The process of identifying a limbo transaction and either committing it or rolling it back is known as *transaction recovery*.

You can attempt to recover **all** limbo transactions or you can perform the recovery, transaction by transaction, using the **ID** of each individual transaction.

Finding limbo transactions

To list the IDs of all limbo transactions, along with an indication of what would happen to each if an automatic two-phase recovery were requested, use the **-l[ist]** switch:



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

gfix -l db_name

Listing limbo transactions and prompting for recovery

Use the **-p[rompt]** switch together with **-l[ist]** to have gfix list the limbo transactions one by one and prompt you for COMMIT or ROLLBACK action:

gfix -l -p db_name

Requesting automated two-phase recovery

The **-t[wo_phase] {ID | all}** switch initiates an automated two-phase recovery.

Use the **all** parameter to perform a two-phase recovery for all limbo transactions:

gfix -t all db_name

Use the **ID** switch by entering the ID of a single transaction for which you want a two-phase recovery performed:

gfix -t ID db_name

where **ID** is the ID of the targeted transaction.

Specifically committing or rolling back limbo transactions

To attempt to resolve limbo transactions by committing them, use the **-c[ommit] {ID | all}** switch. To recover all limbo transactions in this manner, enter:

gfix -c all db_name

To resolve a single limbo transaction by attempting to commit it, enter:

gfix -c ID db_name

where **ID** is the ID of the targeted transaction.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

To attempt to resolve limbo transactions by rolling them back, use the `-r[ollback] {ID | all}` switch. To recover all limbo transactions in this manner, enter:

```
gfix -r all db_name
```

To resolve a single limbo transaction by attempting to roll it back, enter:

```
gfix -r ID db_name
```

where `ID` is the ID of the targeted transaction.

Data validation and repair

In day-to-day operation, a database is sometimes subjected to events that pose minor problems to database structures. These events include:

- *Abnormal termination of a database application*

The integrity of the database is not affected by an abnormal termination. When an application is canceled, committed data is preserved, and uncommitted changes are rolled back. However, if Firebird has already assigned a data page for the uncommitted changes, the orphan page occupies unassigned disk space that should be returned to free space.

- *Write errors in the operating system or hardware*

Write errors usually create problems for database integrity. They can cause data structures such as database pages and indexes to become broken or lost. These corrupt data structures can make committed data unrecoverable.

When to validate a database

You should validate a database whenever

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

- a database backup is unsuccessful
- an application receives a “corrupt database” error
- you suspect data corruption

The command-line utility **gbak** can be used in conjunction with **gfix** to perform a sequence of validation and repair steps.

Database validation is also recommended as a periodic housekeeping task, even if corruption is not suspected, to monitor for and clean up corrupt data structures or misallocated space.

Performing a database validation

Database validation requires exclusive access to the database. Shut down a database to acquire exclusive access. If you do not have exclusive access to the database, you get the error message:

```
OBJECT database_name IS IN USE
```

To validate a database:

```
gfix -v
```

Important Even if you can restore a mended database that reported checksum errors, the extent of data loss may be difficult to determine. If this is a concern, you may want to locate an earlier backup copy and restore the database from it.

Database repair How-to

Get exclusive access

Database validation requires exclusive access to the database. A database must be shut down in order for the SYSDBA or the owner to acquire exclusive access. This error message will appear if you do not have exclusive access:



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

OBJECT database_name IS IN USE

The same message may appear if you are the only user but have another transaction in progress. The **isql** utility, for example, uses up to three concurrent transactions, some of which will not be committed or rolled back until you submit a QUIT or EXIT command to that shell.

- ❖ For information about getting exclusive access, see [Getting exclusive access to a database](#) on p. 283 of chapter 14.

Make a working file copy

Even if you can restore a mended database that reported checksum errors, the extent of data loss may be difficult to determine. If this is a concern, you may want to locate an earlier backup from which you can retrieve missing data after validating and repairing corrupt structures in your current database.

- ❖ Always make sure you work with a *operating system copy* (not a **gbak** file) of your production database. Do not use the **gbak** command, because it cannot back up a database containing corrupt data.

Make sure you have exclusive access to the database when making this copy.

For example, on Windows:

```
copy phoenix.gdb phoenix_copy.gdb
```

and work with **phoenix_copy.gdb**.

Perform the validation

The switches **-v[alidate]** and **-f[ull]** are used to check record and page structures. This checking process reports corrupt structures and releases unassigned record fragments or "orphan pages", i.e. pages that are allocated but unassigned to any data structures.

```
gfix -v -full phoenix_copy.gdb
```

The switch **-n[o_update]** can be used with **-v**, to validate and report on corrupt or misallocated structures without attempting to fix them:



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

```
gfix -v -n phoenix_copy.gdb
```

The **-i[gnore]** switch instructs **gfix** to ignore checksum errors during the validation.

Mend corrupted pages

If **gfix** validation reports damage to data, the next step is to mend (or repair) the database by putting those structures out of the way.

The **-m[end]** switch marks corrupt records as unavailable, so they will be skipped during a subsequent backup. Include a **-f[ull]** switch to request **mend** for all corrupt structures and an **-i[gnore]** switch to bypass checksum errors during the **mend**.

```
gfix -mend -full -ignore phoenix_copy.gdb
```

or, briefly

```
gfix -m -f -i phoenix_copy.gdb
```

Validating after -mend

After the **-mend** command completes its work, again do

```
gfix -v -full phoenix_copy.gdb
```

which will check whether any corrupt structures remain.

Cleaning and recovering the database

Even if errors are still being reported, you should now do a full backup and restore using the **gbak** command-line utility. In its simplest form, the backup command would be

```
gbak -backup -v -ignore phoenix_copy.gdb phoenix_copy.gbk
```



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Dealing with gbak complications during backup

Garbage collection problems: gbak might fail because of problems with garbage collection. If this happens, try again using the following command switches:

```
gbak -backup -v -ignore -garbage phoenix_copy.gdb phoenix_copy.gbk
```

Corruption in limbo record versions: if record versions of a limbo transaction are corrupted, you may need to include the -limbo switch:

```
gbak -backup -v -ignore -garbage -limbo phoenix_copy.gdb phoenix_copy.gbk
```

Restoring the cleaned backup as a new database

Now create a new database, with validation, from the backup using

```
gbak -create -v phoenix_copy.gbk new_phoenix.gdb
```

Verify that restoring the database fixed the problems by validating the restored database with the **-n[o_update]** switch:

```
gfix -v -n phoenix_reborn.gdb
```

If there are problems on restore, you may need to consider further attempts using other gbak switches to eliminate the sources of these problems. For example:

- the **-i[nactive]** switch will eliminate problems with damaged indexes, by restoring without activating any indexes. Afterwards, you can activate the indexes manually, one at a time, until the problem index is found.
- the **-on[e_at_a_time]** switch will restore and commit each table, one by one, allowing you restore good tables and bypass the problem ones.

How to proceed if problems remain

If the steps above do not work, but you are still able to access the corrupt database, you may still be able to transfer table structures and data from the damaged database to a new one, using the **QLI tool**.



Symbols [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

For information about how to attempt this, refer to the documentation pages of the IBPhoenix web site.

Garbage collection and sweeping

Firebird's multi-generational architecture creates the situation where multiple versions of data rows are stored directly on the data pages. Firebird keeps the old versions when a row is updated or deleted, causing the database file(s) to tend to grow out of proportion to the size of accessible data.

Garbage collection

Firebird performs *garbage collection* in the background to limit this growth, by freeing up space allocated to outdated versions of a row whenever it is free of unresolved transactions. Versions that were created in transactions that were rolled back, deleted rows and those rows which are seldom touched, escape this garbage collection.

Garbage collection also happens whenever the database is backed up using `gbak`, since the Firebird server has to touch every row during `gbak`'s task.

Sweeping

Sweeping is a systematic way to remove outdated rows from the database and prevent it from growing too large. However, because performance can be affected during a sweep, it can—and should—be tuned to optimize its benefits while minimizing its impact on users. It can be a positive tuning strategy to disable sweeping, monitor the database statistics and perform sweep manually, either on an “as required” basis or at scheduled times.

Exclusive access is not required to perform sweeps.

For information about how database statistics reports can help in the analysis of the sweeping requirements in your database, see [gstat command-line tool](#) on p. 452 of chapter 23.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Sweep Interval

The Firebird server maintains an inventory of transactions. Any transaction that is uncommitted is known as an *interesting transaction*. The oldest of these “interesting” transactions (Oldest Interesting Transaction—OIT) marks the starting point for the sweep interval. If the sweep interval setting is greater than zero, Firebird initiates a full sweep of the database when the difference between the OIT and the newest transaction passes the threshold set by the sweep interval.

Considerations for sweeping

- Sweeping a database can affect transaction start-up if rolled back transactions exist in the database. As the time since the last sweep increases, the time for transaction start-up can also increase. Lowering the sweep interval can help reduce the time for transaction start-up.
- On the obverse side of the coin, too-frequent database sweeps can reduce application performance. Raising the sweep interval could help improve overall performance.
- Unless the database contains many rolled back transactions, changing the sweep interval has little effect on database size.

Default sweep interval

The default sweep interval is 20 000 transactions.

★ It is a subtle but important distinction that the automatic sweep does *not* necessarily occur every 20,000 transactions. It is only when the *difference* between the OIT and the newest transaction reaches the threshold. If every transaction to the database is committed promptly, then this difference it is not likely to be great enough to trigger the automatic sweep.

Setting the sweep interval

The option switch is **-h[ousekeeping] n**, where **n** represents the count (interval) which you want to change to. To set the automatic sweep threshold to **n** transactions:

```
gfix -h n db_name
```

Disabling automatic sweeping

Automatic sweeping can be *disabled* by setting a sweep interval of zero. You might consider disabling the automatic sweep if you need to maximize throughput in a high-input database and need to avoid the occasional delays imposed by sweeps.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

To disable automatic sweeping, enter this command:

```
gfix -h 0 db_name
```

This is only recommended if database housekeeping is being managed effectively by an alternative means, viz.. one or a combination of

- frequent backups
- systematic monitoring of the difference between the OIT and the newest transaction closely
- scheduled *manual sweeps* during quiet times

Performing a manual sweep

A manual sweep can be done at any time to release space held by back versions—records that were rolled back or made obsolete by deletions or updates. It is common to schedule sweeps at a time of least activity on the database server, to avoid competing with clients for resources.

If you are watching the interval between the OIT and newest transaction frequently and want to look after sweeps yourself, or if a large run of deletions has been done, or if occasional updates have built up a long interval and are causing performance to slow down, you may wish to *perform a manual sweep* using the **-sweep** switch:

```
gfix -sweep db_name
```

This initiates a sweep immediately.

Exclusive access for manual sweeps

Sweeping a database does not strictly require it to be shut down—it can be done at any time—but it can impact system performance and should not be done at busy times.

There is a benefit if a sweep is performed with exclusive access and there are no outstanding active transactions, which will reduce memory use and improve performance. Under these conditions, the sweep will update the state of data records and the state of the inventory of past transactions. Non-committed transactions are finally rendered obsolete, obviating the need for internal data structures to keep tracking them in order to maintain snapshots of database versions.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Shadows

 For details about creating and using database shadows, see [Database shadows](#) on page 379 in chapter 20.

Database name is not used in the commands pertaining to shadow databases.

Activating a shadow

The switch for activating a shadow when a database dies is **-ac[**tivate**]**.

Syntax

```
gfix -ac
```

Dropping unavailable shadows

The switch for dropping unavailable shadows is **-k[**ill**]**.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Syntax

gfix -k

Summary of gfix switches

TABLE 24-1 Summary of gfix switches

Switch	Task	Description
-at[tach] <i>n</i>	Shutdown	Used with -shut to prevent new database connections during timeout period of <i>n</i> seconds. Shutdown will be cancelled if there are still active connections after <i>n</i> seconds.
-b[uffers] <i>n</i>	Cache buffers	Set default database cache buffers for the database to <i>n</i> pages. This is the recommended way to set the default database cache size.
-ca[che] <i>n</i>		Not used
-c[ommit] { <i>ID</i> all}	Transaction recovery	Commit limbo transaction specified by <i>ID</i> or commit all limbo transactions.
-f[orce] <i>n</i>	Shutdown	Used with -shut to force shutdown of a database after <i>n</i> seconds—a drastic solution that should be used only as a last resort.
-full	Data repair	Used with -v[alidate] to check record and page structures, causing unassigned record fragments to be released.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

TABLE 24-1 Summary of gfix switches

Switch	Task	Description
-h[ousekeeping]	Sweeping	Change threshold for automatic sweeping to n transactions. Default is 20,000. Set n to 0 to disable automatic sweeping.
-i[ggnore]	Data repair	Ignore checksum errors when validating or sweeping.
-l[ist]	Transaction recovery	Display IDs of each limbo transaction and indicate what would occur if -t[wo_phase] were used for automated two-phase recovery.
-m[end]	Data repair	Mark corrupt records as unavailable, so that they will be skipped during a subsequent validation or backup
-n[o_update]	Data repair	Used with -v[alidate] to validate corrupt or misallocated structures, reporting them but not fixing them.
-o[nline]	Shutdown	Cancels a -shut operation that has been seheduled, or rescinds a shutdown that is currently in effect.
-pa[sword] <i>password</i>	Remote access	Submits <i>password</i> for accessing database. For most gfix operations, this must be the password of the SYSDBA, the database owner or (on Linux/UNIX) a user with <i>root</i> privileges.
-p[rompt]	Transaction recovery	Used with -l[ist] to prompt for action during transaction recovery
-r[ollback] { <i>ID</i> all}	Transaction recovery	Roll back limbo transaction specified by <i>ID</i> or roll back all limbo transactions.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

TABLE 24-1 Summary of gfix switches

Switch	Task	Description
<code>-s[weep]</code>	Sweeping	Force an immediate sweep of the database. Usually only useful where automatic sweeping is disabled.
<code>-sh[ut]</code>	Shutdown	Shut down the database. Requires to be qualified by either <code>-at[ach]</code> , <code>-f[orce]</code> or <code>-tr[an] n</code> .
<code>-t[wo_phase] {ID all}</code>	Transaction recovery	Perform automated two-phase recovery, either for a limbo transaction specified by <i>ID</i> or for all limbo transactions.
<code>tr[an] n</code>	Shutdown	Used with <code>-shut</code> to prevent new transactions from starting during timeout period of <i>n</i> seconds. Shutdown will be cancelled if there are still active transactions after <i>n</i> seconds.
<code>-use {reserve full}</code>	Use all space	Enable or disable full use of the space available on database pages. The default <i>reserve</i> fills pages using a fill ratio of 80 percent. Switching to <i>full</i> uses all space available.
<code>-user username</code>	Remote access	Submits <i>username</i> for accessing database. For most gfix operations, this must be SYSDBA, the database owner or (on Linux/UNIX) a user with <i>root</i> privileges.
<code>-v[alidate]</code>	Data repair	Locate and release pages that are allocated but unassigned to any data structures. Also reports corrupt structures.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

TABLE 24-1 Summary of **gfix** switches

Switch	Task	Description
-w[rite] {sync async}	Forced writes	Enable or disable forced (synchronous, buffered) writes. sync enables, async disables.
-z		Report version of gfix and Firebird server.

gfix error messages

TABLE 24-2 **gfix** error messages

Error Message	Causes and Suggested Actions to Take
Database file name <string> already given	A command-line option was interpreted as a database file because the option was not preceded by a hyphen (-) or slash (/). Correct the syntax.
Invalid switch	A command-line option was not recognized.
Incompatible switch combinations	You specified at least two options that do not work together, or you specified an option that has no meaning without another option (for example, -full on its own).
More limbo transactions than fit. Try again.	The database contains more limbo transactions than gfix can print in a single session. Commit or roll back some of the limbo transactions, then try again.
Numeric value required	The -housekeeping option requires a single, non-negative argument specifying number of transactions per sweep.
Please retry, specifying <string>	Both a file name and at least one option must be specified.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

TABLE 24–2 gfix error messages

Error Message	Causes and Suggested Actions to Take
Transaction number or "all" required	You specified <code>-commit</code> , <code>-rollback</code> , or <code>-two_phase</code> without supplying the required argument.
<code>-mode read_only</code> or <code>read_write</code>	The <code>-mode</code> option takes either <code>read_only</code> or <code>read_write</code> as a switch.
" <code>read_only</code> " or " <code>read_write</code> " required	The <code>-mode</code> option must be accompanied by one of these two arguments.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

CHAPTER 25

Programming on Firebird Server

One of the great benefits of a full-blooded SQL relational database implementation is its capability to compile and execute internal code modules—stored procedures and triggers—supplied as source code by the developer. The language that provides this capability on a Firebird server is PSQL—a simple but powerful set of SQL extensions that combines with regular data manipulation language (DML) statements to become compilable source modules.



OTHER REFERENCES

A complete description of the elements of PSQL can be found in *Firebird Reference Guide*—[PSQL-Firebird Procedural Language](#) (ch. 3 p. 222).

Overview of PSQL code modules

The high-level language for Firebird server-side programming is SQL. Source code is presented to the engine in the form of SQL programming language extensions—PSQL statements and constructs—and DML statements. These statements are, themselves, wrapped inside single DDL statements. The objective of each of these “DDL super-statements” is to create and store a new executable code object (stored procedure or trigger) or to redefine (ALTER) an existing object. A DDL statement is also used to destroy (DROP) executable code objects.

PSQL supports the three data manipulation statements: INSERT, UPDATE, DELETE and, for stored procedures, the ability to select a single-row set of data items into a set of host variables—the singleton SELECT. The PSQL extensions provide the following capabilities:

- local variables and assignment statements
- conditional control-flow statements
- context variables for triggers, storing the old and new values of each column of all DML input sets
- posting of user-defined database events to the listening client

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

- exceptions, including user-defined exceptions, and mechanisms for error handling
- input and output arguments
- awkward cursor declarations and fetching characteristics encapsulated in *FOR SELECT..DO* looping mechanism for stored procedures
- the SUSPEND statement, which allows the capability to write stored procedures which output directly to the input list of a SELECT statement without intermediate client calls—*selectable stored procedures*
- embedding of stored procedure calls in both stored procedures and triggers
- capability to define multiple BEFORE and AFTER triggers for each DML event and to position them in a predefined execution order

Except for the special-purpose elements mentioned, the entire PSQL language set is available to both stored procedures and triggers.

Restrictions on PSQL

- Statements using the data definition language (DML) subset of Firebird SQL are not permissible in PSQL. However, DDL statements can be batched into linkable scripts and submitted to the server for processing using the `isql` utility or a program providing `isql` capability—see chapter 12, [Writing and Running Scripts](#) (p. 222).
- Transaction control statements are not valid in PSQL, because stored procedures and triggers always execute within an existing client transaction context and Firebird does not support embedded transactions
- Some other statement types are reserved for use in different environments, e.g. scripts or embedded SQL. For a full list of excluded statements, see [Statement types not supported in PSQL](#) on page 496.
- Metadata objects cannot be passed to nor returned from stored procedures as arguments.
- Trigger procedures cannot accept or return arguments

Elements of procedures and triggers

Stored procedures and triggers are defined with the CREATE PROCEDURE and CREATE TRIGGER statements, respectively. Each of these statements is composed of a *header* and a *body*.



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

Header elements

- The name of the procedure or trigger, unique within the database.
- For a trigger:
 - A table name, identifying the table that causes the trigger to fire.
 - a parameter that determines *when* the trigger fires.
- For a stored procedure:
 - An optional list of *input parameters* and their datatypes.
 - If the procedure returns values to the calling program, a list of *output parameters* and their datatypes.

Body elements

- An optional list of *local variables* and their datatypes.
- A *block* of statements in Firebird procedure and trigger language, bracketed by BEGIN and END. A block can itself include other blocks, so that there may be many levels of nesting.

Statement types not supported in PSQL

PSQL excludes all DML statements and also some which are reserved for use with embedded SQL (ESQL). Because stored procedures always run within a client transaction context, the transaction control statements are excluded, as are the security management statements.

The following statement types are not supported in triggers or stored procedures:

- Data definition language statements: CREATE, ALTER, DROP, SET GENERATOR, DECLARE EXTERNAL FUNCTION, and DECLARE FILTER
- Transaction control statements: SET TRANSACTION, COMMIT, ROLLBACK
- Dynamic SQL statements: PREPARE, DESCRIBE, EXECUTE
- CONNECT/DISCONNECT, and sending SQL statements to another database
- GRANT/REVOKE
- EVENT INIT/EVENT WAIT

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

- BEGIN DECLARATION/END DECLARE SECTION
- BASED ON
- WHENEVER
- DECLARE CURSOR
- OPEN
- FETCH

Language elements

The following table shows the PSQL language elements available in Firebird:

TABLE 25-1 PSQL extensions for stored procedures and triggers

Statement	Description
BEGIN ... END	Defines a block of statements that executes as one; the BEGIN reserved word starts the block, the END reserved word terminates it. Neither should be followed by a semicolon.
<i>variable</i> = <i>expression</i>	Assignment statement which assigns the value of <i>expression</i> to <i>variable</i> , a local variable, input parameter, or output parameter
/* <i>comment_text</i> */	Programmer's comment, where <i>comment_text</i> can be any number of lines of text
EXCEPTION <i>exception_name</i>	Raises the named exception Exception: A user-defined error that can be handled with WHEN



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

TABLE 25-1 PSQL extensions for stored procedures and triggers (*continued*)

Statement	Description
EXECUTE PROCEDURE <i>proc_name</i> [<i>var</i> [, <i>var</i> ...]] [RETURNING_VALUES <i>var</i> [, <i>var</i> ...]]	Executes stored procedure, <i>proc_name</i> , with the input arguments listed following the procedure name, returning values in the output arguments listed following RETURNING_VALUES Enables nested procedures and recursion Input and output parameters must be variables defined within the procedure.
EXIT	Jumps to the final END statement in the procedure
FOR <i>select_statement</i> DO <i>compound_statement</i>	Repeats the statement or block following DO for every qualifying row retrieved by <i>select_statement</i> <i>select_statement</i> : a normal SELECT statement, except that the INTO clause is required and must come last
<i>compound_statement</i>	Either a single statement in procedure and trigger language or a block of statements bracketed by BEGIN and END
IF (<i>condition</i>) THEN <i>compound_statement</i> [ELSE <i>compound_statement</i>]	Tests <i>condition</i> and if it is TRUE, performs the statement or block following THEN. Otherwise, performs the statement or block following ELSE, if present. <i>condition</i> : a Boolean expression (TRUE, FALSE, or UNKNOWN), generally two expressions as operands of a comparison operator.
NEW. <i>column</i>	New context variable that indicates a new column value in an INSERT or UPDATE operation. For use in triggers only, but available in a CHECK constraint.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

TABLE 25-1 PSQL extensions for stored procedures and triggers (*continued*)

Statement	Description
OLD. <i>column</i>	Old context variable that indicates a column value before an UPDATE or DELETE operation. For use in triggers only, but available in a CHECK constraint.
POST_EVENT <i>event_name</i>	Posts the event <i>event_name</i> .
SUSPEND	In a SELECT procedure: <ul style="list-style-type: none">• Suspends execution of procedure until next FETCH is issued by the calling application• Returns output values, if any, to the calling application.• Not recommended for executable procedures.
WHILE (<i>condition</i>) DO <i>compound_statement</i>	While <i>condition</i> is TRUE, keep performing <i>compound_statement</i> . First <i>condition</i> is tested, and if it is TRUE, then <i>compound_statement</i> is performed. This sequence is repeated until <i>condition</i> is no longer TRUE.
WHEN { <i>error</i> [, <i>error</i> ...] ANY} DO <i>compound_statement</i>	Error-handling statement. When one of the specified errors occurs, performs <i>compound_statement</i> . WHEN statements, if present, must come at the end of a block, just before END. <ul style="list-style-type: none">• <i>error</i>: EXCEPTION <i>exception_name</i>, SQLCODE <i>errcode</i> or GDSCODE <i>number</i>.• ANY: Handles any errors.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Coding the body of the code module

The CREATE statement

Source code for procedures and triggers is constructed inside a “super-statement” which begins with the keywords CREATE PROCEDURE or CREATE TRIGGER and ends with a terminator symbol following the final END statement. For example,

```
CREATE PROCEDURE Name...  
...  
AS  
...  
BEGIN  
...  
END ^
```

With both stored procedures and triggers, all statements following the keyword AS comprise the statements which lay out the program logic. The main difference between triggers and stored procedures is in the **header portion** of the CREATE statement.

BEGIN...END blocks

PSQL is a structured language. Once variables are declared, the procedural statements are bounded by the keywords BEGIN and END. In the course of developing the logic of the procedure, other blocks can be embedded; and any block can embed another block. Each embedded block of statements is bounded by a BEGIN...END pair.

No terminator symbol is used for the BEGIN or END keywords, except for the final END keyword that closes the procedure block and terminates the CREATE PROCEDURE or CREATE TRIGGER statement. This final END keyword takes the special terminator that was defined with SET TERM before the CREATE PROCEDURE or CREATE TRIGGER directive.



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

Statement terminator

Each statement in a stored procedure or trigger body must be terminated by a semicolon. Because each of the source code statements of your procedure is, itself, an element of a single CREATE PROCEDURE or CREATE TRIGGER statement, it is necessary to define a different symbol to terminate the CREATE PROCEDURE statement in ISQL.

- Use SET TERM before CREATE PROCEDURE or CREATE TRIGGER to specify a terminator other than a semicolon. It can be any string symbol you like, except `;'. e.g.

```
SET TERM ^ ;
```

The terminator can be a multi-character string, if you prefer. The string should not be a keyword and it can contain embedded blanks. A blank on its own will not work.

- Terminate the last END statement of the procedure definition using the alternative terminator you set:

```
...
```

```
END ^
```

- After the final END statement that terminates the CREATE PROCEDURE or CREATE TRIGGER statement, include another SET TERM to change the terminator back to a semicolon, e.g. to switch back from the example above:

```
SET TERM ; ^.
```

The following example illustrates the use of SET TERM for a trigger. The terminator is temporarily set to a double exclamation point (!!).



Symbols **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

```
SET TERM !! ;
CREATE TRIGGER SIMPLE FOR EMPLOYEE
AFTER UPDATE AS
BEGIN
...
END !!
SET TERM ; !!
```

Using variables

Four types of variables can be used in the body of a code module, with some restrictions depending on whether the module is a stored procedure or a trigger:

- local variables, used to hold values used only within the trigger or procedure.
- the NEW.ColumnName and OLD.ColumnName context variables, restricted to use in triggers, which store the new and old values of each column of a table when a DML operation is pending.
- input parameters, used to pass values from an application to a stored procedure. Not available for triggers.
- output parameters, used to pass values from a stored procedure back to the calling application. Not available for triggers.

Any of these types of variables can be used in the body of a stored procedure where an expression can appear. They can be assigned a literal value, or assigned a value derived from queries or expression evaluations.

The colon (:) marker for variables

In SQL statements, prefix the variable's name with a colon character (:) whenever

- the variable is receiving a value from a SELECT...INTO construct and the variable name matches the column name. For example,

```
SELECT COL1, COL2, COL3 FROM TABLEB
...
INTO :COL1, :COL2, :COL3;
```

- The variable's value is being used in an SQL statement:

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

- in the VALUES clause of an INSERT statement, e.g.

```
INSERT INTO TABLEA (COL1, COL2)
```

```
VALUES (:invar1, :invar2);
```

- as the value to which a column is updated, using SET, e.g.

```
UPDATE TABLEA
```

```
SET COL1 = :invar1;
```

- as the right side of a WHERE predicate *unless* the variable is a NEW or OLD context variable. For example:

```
SELECT * FROM TABLEA
```

```
WHERE COL2 = :invar2;
```

Omit the colon in all other situations.

- Never prefix context variables with a colon..

Local variables

Local variables are declared and used within a stored procedure. They have no effect outside the procedure.

Local variables must be declared at the beginning of a procedure body before they can be used.

❖ Always initialize local variables as early as possible in the procedure body.

Input parameters

Not available to triggers.

Input parameters are used to pass values from an application to a procedure. They are declared in a comma-delimited list in parentheses following the procedure name. Once declared, they can be used in the procedure body anywhere an expression can appear.

Input parameters are passed *by value* from the calling program to a stored procedure. This means that if the procedure changes the value of an input parameter, the change has effect only within the procedure. When control returns to the calling program, the input parameter still has its original value.

Output parameters

Not available to triggers.

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

The output parameter list is used to specify the values that are to be returned from a procedure to the calling application or procedure. Declare them in a comma-delimited list in parentheses following the RETURNS reserved word in the procedure header. Once declared, they can be used in the procedure body anywhere an expression can appear.

For example, the following procedure header declares five output parameters, HEAD_DEPT, DEPARTMENT, MNGR_NAME, TITLE, and EMP_CNT:

```
CREATE PROCEDURE ORG_CHART
RETURNS (HEAD_DEPT CHAR(25), DEPARTMENT CHAR(25),
MNGR_NAME CHAR(20), TITLE CHAR(5), EMP_CNT INTEGER)
```

If you declare output parameters in the procedure header, the procedure must assign values to them to return to the calling application. Values can be derived from any valid expression in the procedure.

- ❖ Always initialize output parameters before beginning to process the data which will be sent to them.

NEW and OLD context variables

Not available to stored procedures.

Triggers can use two context variables, OLD, and NEW. The OLD context variable refers to the current or previous values in a row being updated or deleted. OLD is not used for inserts. NEW refers to a new set of INSERT or UPDATE values for a row. NEW is not used for deletes. Context variables are often used to compare the values of a column before and after it is modified.

The syntax for context variables is as follows:

```
NEW.column
OLD.column
```

where *column* is any column in the affected row. Context variables can be used anywhere a regular variable can be used.

New values for a row can only be altered by *before* actions. A trigger that fires after INSERT and tries to assign a value to NEW.*column* will have no effect. The actual column values are not altered until after the action, so triggers that reference values from their target tables will not see a newly inserted or updated value unless they fire after UPDATE or INSERT.

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

For example, the following trigger fires after the EMPLOYEE table is updated, and compares an employee's old and new salary. If there is a change in salary, the trigger inserts an entry in the SALARY_HISTORY table.

```
SET TERM !! ;
CREATE TRIGGER SAVE_SALARY_CHANGE FOR EMPLOYEE
AFTER UPDATE AS
```

★ Since Firebird creates triggers to implement CHECK constraints, the OLD and NEW context variables can be used directly in a CHECK constraint, for example:

```
ALTER TABLE employee
ADD CONSTRAINT employee_salary_raise_ck
CHECK ((old.salary IS NULL) OR (new.salary >= old.salary));
```

Assignment statements

A procedure can assign values to variables with the syntax:

```
variable = expression;
```

where *expression* is any valid combination of variables, operators, and expressions, and can include sub-selects and calls to user-defined functions (UDFs), SQL functions and generators.

Variables should be assigned values of the data type that they are declared to be. Numeric variables should be assigned numeric values, and character variables assigned character values. Although Firebird provides automatic type conversion in some cases, it is advisable to use explicit casting to avoid unanticipated mismatches.

For more information about explicit casting and data type conversion, see [The CAST\(\) function](#) on page 254 in chapter 13.



Symbols **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

```
BEGIN  
IF (old.salary <> new.salary) THEN  
INSERT INTO SALARY_HISTORY (EMP_NO, CHANGE_DATE,  
UPDATER_ID, OLD_SALARY, PERCENT_CHANGE)  
VALUES (old.emp_no, 'now', USER, old.salary,  
(new.salary - old.salary) * 100 / old.salary);  
END !!  
SET TERM ; !!
```

Using SELECT...INTO statements

In a stored procedure, use the SELECT statement with an INTO clause to retrieve one or more column values from the tables and assign them to host variables.

Singleton SELECTS

An ordinary SELECT statement in PSQL must return at most one row from the database—a standard singleton SELECT. An ISC error will occur if the statement returns more than one row. An ORDER BY clause is not valid for a singleton select.

Normal rules apply to the input list and the WHERE and GROUP BY clauses, if used. The INTO clause is required and must be the last clause in the statement.

For example, the following is a singleton SELECT statement in a parameterized DSQL query in an application:

```
SELECT SUM(BUDGET), AVG(BUDGET)  
FROM DEPARTMENT  
WHERE HEAD_DEPT = :head_dept;
```

To use this SELECT statement in a procedure, add local variables or output parameters and add the INTO clause to the end as follows:

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

```
SELECT SUM(BUDGET), AVG(BUDGET)
FROM DEPARTMENT
WHERE HEAD_DEPT = :head_dept
INTO :tot_budget, :avg_budget;
```

 For more details about SELECT statement syntax, see [The SELECT query](#) on p. 117 of chapter 9 and the relevant entry in the SQL Statement and Function Reference section of the *Firebird Reference Guide*

The WHILE...DO construct

WHILE...DO is a looping construct that repeats a statement or block of statements as long as a condition is true. The condition is tested at the start of each loop. WHILE...DO uses the following syntax:

```
WHILE (<condition>) DO
<compound_statement>
<compound_statement> =
{<block> | statement; }
```

The *compound_statement* is executed as long as *condition* remains TRUE.

A *block* is one or more compound statements enclosed by BEGIN and END.

For example, the following procedure uses a WHILE...DO loop to compute the sum of all integers from one up to the input parameter:

```
SET TERM !! ;
CREATE PROCEDURE SUM_INT (I INTEGER) RETURNS (S INTEGER)
AS
BEGIN
  S = 0;
```

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

```
WHILE (i > 0) DO  
BEGIN  
  s = s + i;  
  i = i - 1;  
END  
END!!
```

If this procedure is called from **isql** with the command:

```
EXECUTE PROCEDURE SUM_INT 4;
```

then the results will be:

```
S  
=====
```

10

The IF...THEN...ELSE construct

The IF...THEN...ELSE construct branches to alternative courses of action by testing a specified condition. The syntax of IF...THEN...ELSE is:

```
IF (<condition>)  
THEN <compound_statement>  
[ELSE <compound_statement>]  
<compound_statement> = {<block> | statement; }
```

The *condition* clause is a predicate that must evaluate to TRUE to execute the statement or block following THEN. The optional ELSE clause specifies an alternative statement or block to be executed if *condition* is FALSE.

- ❖ The predicate being tested by IF must be enclosed in brackets.

The following lines of code illustrate the use of IF...THEN, assuming the variables LINE2, FIRST, and LAST have been previously declared:



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

...
IF (FIRSTNAME IS NOT NULL)
THEN LINE2 = FIRSTNAME || ' ' || LASTNAME;
ELSE LINE2 = LASTNAME;

...

❖ Pascal programmers—observe that the IF...THEN branch is terminated!

Using SUSPEND, EXIT, and END

The SUSPEND statement suspends execution of a select procedure, passes control back to the program, and resumes execution from the next statement when the next FETCH is executed. SUSPEND also returns values in the output parameters of a stored procedure.

❖ SUSPEND should not be used in executable procedures or triggers, since the statements that follow it will never execute. Use EXIT instead to indicate to the reader explicitly that the statement terminates the procedure.

In a select procedure, the SUSPEND statement returns current values of output parameters to the calling program and continues execution.

❖ If an output parameter has not been assigned a value, its value is unpredictable, which can lead to errors. A procedure should ensure that all output parameters are initialized in advance of the processing that will assign values, to ensure that a SUSPEND will pass valid output.

In both select and executable procedures, EXIT causes program control to jump to the final END statement in the procedure.

What happens when a procedure reaches the final END statement depends on the type of procedure:

- In a select procedure, the final END statement returns control to the application and sets SQLCODE to 100, which indicates there are no more rows to retrieve.
- In an executable procedure, the final END statement returns control and values of output parameters, if any, to the calling application.
-

The behaviors of these statements are summarized in the following table:



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

:::

TABLE 25-2 SUSPEND, EXIT, and END

Procedure type	SUSPEND	EXIT	END
Select procedure	Suspends execution of procedure until next FETCH Returns values	Jumps to final END	Returns control to application Sets SQLCODE to 100
Executable procedure	Jumps to final END Not Recommended	Jumps to final END	Returns values Returns control to application
Triggers	Never used	Jumps to final END	Causes execution of the next trigger in the same phase (BEFORE or AFTER) as the current one, if any; otherwise terminates trigger processing for the phase..

Flow of control effects

Example

Consider the following procedure:

```
SET TERM !!;  
CREATE PROCEDURE P RETURNS (R INTEGER)  
AS  
BEGIN  
R = 0;  
WHILE (R < 5) DO  
BEGIN  
R = R + 1;
```

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

```
SUSPEND;  
IF (R = 3) THEN  
EXIT;  
END  
END;  
SET TERM ;!!
```

If this procedure is used as a select procedure, for example:

```
SELECT * FROM P;
```

then it returns values 1, 2, and 3 to the calling application, since the SUSPEND statement returns the current value of R to the calling application. The procedure terminates when it encounters EXIT.

If the procedure is used as an executable procedure, for example:

```
EXECUTE PROCEDURE P;
```

then it returns 1, since the SUSPEND statement terminates the procedure and returns the current value of R to the calling application. ♦♦ This is not recommended ♦♦

Statements following SUSPEND

If a select procedure has executable statements following the last SUSPEND in the procedure, all of those statements are executed, even though no more rows are returned to the calling program. The procedure terminates with the final END statement.

Error behavior

When a procedure encounters an error—either a SQLCODE error, GDSCODE error, or user-defined exception—all statements since the last SUSPEND are undone.

- Because select procedures can have multiple SUSPENDs, possibly inside a loop statement, only the actions since the last SUSPEND are undone.
- Since executable procedures should not use SUSPEND, when an error occurs the entire executable procedure is undone (if EXIT is used, as recommended).

For more information about handling errors in PSQL, see [Handling exceptions](#) on page 552.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Using generators

In a read-write database, a *generator* is a database object that automatically increments each time the special function, GEN_ID(), is called. For more information, see [Firebird Generators](#) on p. 296 of chapter 15.

❖ Generators cannot be used in read-only databases.

Generators are particularly useful in triggers for inserting unique column values, for example, primary key values. GEN_ID() is a function call that can be used in a statement anywhere that a constant value can be used.

❖ Ensure that the generator exists before using it in a trigger or stored procedure.

The following trigger uses GEN_ID() to increment a new customer number before values are inserted into the CUSTOMER table. Because it assigns a value to NEW.CUST_NO., it must be a BEFORE INSERT trigger. The NEW context variable has no effect in AFTER triggers.

```
:SET TERM !! ;
CREATE TRIGGER SET_CUST_NO FOR CUSTOMER
BEFORE INSERT AS
BEGIN
  IF (NEW.CUST_NO IS NULL) THEN
    NEW.CUST_NO = GEN_ID(CUST_NO_GEN, 1);
  END !!
SET TERM ; !!
```

Event alerters

Firebird events provide a signaling mechanism which enables applications to listen for database changes made by concurrently running applications without the need applications to incur CPU cost or consume bandwidth to poll one another directly.

On the server side, it is necessary to set up an *event alerter* in a trigger or stored procedure to post an event signal. When the transaction commits, any events that occurred can be posted to a listening client application. The application can then respond to the event by, for example, refreshing its open datasets upon being alerted to changes.

**Symbols** [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

For details of how to implement listening for events on clients, refer to [Handling events on a client](#) at the end of this chapter.

To use an event alerter in a stored procedure or trigger, use the following syntax:

```
POST_EVENT <event_name>;
```

The parameter, *event_name*, can be either a quoted literal or string variable.

- ❖ A variable name should not be prefixed by a colon in this context.

When the procedure is executed, this statement notifies the event manager, which alerts applications waiting for the named event. For example, the following statement posts an event named "new_order":

```
POST_EVENT 'new_order';
```

Alternatively, using a *variable* for the event name allows a statement to post different events, depending on the value of the string variable, *event_name*.

```
POST_EVENT event_name;
```

- ❖ For more information on events and event alerters:

- using the API and direct-to-API components—see [The AST function](#) on page 564 and the API Guide (APIGuide.pdf) in the InterBase® 6 documentation set, available from Borland.
- in embedded applications, refer to the *Embedded SQL Guide* (EmbedSQL.pdf) in the InterBase® 6 documentation set.

Adding comments

Stored procedure code should be commented to aid debugging and application development. Comments are especially important in stored procedures since they are global to the database and can be used by many different application developers.

- ❖ For details of including comments, see [Comments](#) on page 226 in chapter 12, [Writing and Running Scripts](#).



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Case-sensitivity and white space

If double-quote delimiters were used when your database objects were defined, then all case-sensitivity and quoting rules that apply to your data in dynamic SQL must also be applied when you refer to those objects in procedure statements.

All other code is non case-sensitive. For example (assuming no delimited object identifiers are involved) these two statement fragments are equivalent:

```
CREATE PROCEDURE MYPROC...
create procedure myproc...
```

The compiler places no restrictions on white space or line feeds. It can be useful, for readability of your code, to adopt some form of standard convention in the layout of your procedure code. For example, you might write all keywords in upper case, indent code blocks, list variable declarations in separate lines, place comma separators at the beginning of lines, and so on.

Managing compiled objects

Considering that the high-level language for Firebird server-side programming is SQL and that source code is presented to the engine in the form of DDL “super-statements” for compilation into database objects, it is not surprising that all code maintenance is also performed using DDL statements. These statements are consistent with the conventions for maintenance of other objects in SQL databases:

- Redefinition of compiled objects—stored procedures and triggers—employs the `ALTER <object>` syntax.
★ For stored procedures, Firebird also provides `RECREATE PROCEDURE`.
- Removal of compiled objects from the database employs `DROP` statements.

Managing your code

There are two ways to manage procedures and triggers: entering the statements interactively through `isql` or another tool which can submit direct DSQL; or with an input file containing data definition statements—commonly known as a script.

For details about writing scripts, see chapter 12, [Writing and Running Scripts](#) (p. 222).



Symbols **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

For information about using isql, see chapter 10, [Interactive SQL Utility \(isql\)](#) (p. 152).

- ★ The interactive interface seems a quick and easy way to do things—until the first time you need to change, check or recover something. Using scripts is the recommended way, because scripts not only provide retrievable documentation of the code you submitted, but can include comments and can be modified easily.

Editing tools

Any ASCII plain text editor can be used, as long as it does not save any non-printable characters other than carriage returns (ASCII 13), line feeds (ASCII 10) and tab characters (ASCII 9). Some editors are available which provide syntax highlighting for SQL: the IDE editors of Borland Delphi™ and Kylix™, along with several of the tools listed in the *Firebird Reference Guide*.

Compiling stored procedures and triggers

To complete any script file, you must include at least one “empty line” past the last statement or comment in the file. To do this, press the `[Enter]` key at least once in your text editor.

When you have completed your procedure, save it to a file of any name.

- ★ It is a common practice add the extension “.sql” to Firebird script files. Apart from its usefulness in identifying scripts in your filesystem, the “.sql” extension is recognized as an SQL batch file by many editing tools that support SQL syntax highlighting.

To compile your stored procedure, simply run your script using the INPUT command in isql or through the scripting interface of your database management tool.

Script errors

Firebird generates errors during parsing if there is incorrect syntax in a CREATE PROCEDURE statement. Error messages look similar to this:

**Symbols** [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

Statement failed, SQLCODE = -104

Dynamic SQL Error

-SQL error code = -104

-Token unknown - line 4, char 9

-tmp

The line numbers are counted from the beginning of the CREATE PROCEDURE statement, not from the beginning of the script file. Characters are counted from the left, and the unknown token indicated is either the source of the error, or immediately to the right of the source of the error. When in doubt, examine the entire line to determine the source of the syntax error.

- ★ If you are using a version of Firebird **isql** later than Firebird 1.0, you will notice an improvement in its capability to describe script errors and to pinpoint their locations. Although Firebird does not provide a debugging feature for stored procedures, several third-party tools now provide this capability.

Object dependencies

The Firebird engine is meticulous about maintaining information about the inter-dependencies of the database objects under its management. Furthermore, the engine will defer or disallow compilation of ALTER or DROP statements if the existing compiled version is found to be involved in an *"interesting transaction"*. For a description of this condition, see ["Object is in use" error](#) on page 518.

Privileges required for altering objects

Only SYSDBA and the owner of a procedure or trigger can alter it.

Steps

When you alter a procedure or trigger, the new procedure definition replaces the old one. To alter a procedure or trigger definition, follow these steps:

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

- 1 Copy the original data definition file used to create the procedure. Alternatively, use `isql -extract` to extract a procedure from the database to a file. See [Extracting metadata](#) in the chapter [Interactive SQL Utility \(isql\)](#).
- 2 Edit the file, changing CREATE to ALTER, and changing the procedure definition as desired. Retain whatever is still useful.

Testing during development

During application development, create and test stored procedures interactively with `isql` or in your favorite database desktop utility. Once a stored procedure has been created, tested, and refined, it can be used in applications.

Timing of modifications

You must make special considerations when making changes to stored procedures that are currently in use by other requests. A procedure is in use when it is currently executing, or if it has been compiled internally to the metadata cache by an outstanding DDL statement request.

Changes to procedures are not visible to client applications until they disconnect from and reconnect to the database.

- ❖ Triggers and procedures that invoke altered procedures do not have access to the new version until the database is in a state in which *all* clients are disconnected.

Choosing your time

To simplify the task of altering or dropping stored procedures or triggers, it is highly recommended to perform this task during a maintenance period when no client applications are connected to the database. This ensures that all client applications see the same version of a stored procedure *before* and *after* you make an alteration.

- ★ To reduce the maintenance period, it is possible to perform the alteration while the database is in use, instruct all users to commit their work and then briefly close all client applications. It can thus be said that it is safe to alter procedures whilst the database is in use, since the alterations will be deferred until the Firebird engine determines it is safe to proceed.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

“Object is in use” error

This error tends to frustrate developers more than any other. You are logged in, typically, as SYSDBA or owner of the database. You have exclusive access—which is desirable when changing metadata—and yet, here is this phantom user, somewhere, using the object whose metadata you want to modify or delete.

The source of this mystery will be one or more of the following:

- In shutting down the database, preparing to get exclusive access, you (or another human) were already logged in as SYSDBA, owner or (on Linux/UNIX) a suitably privileged OS user. In testing the conditions for shutdown, Firebird ignores these users and any transactions they may have current, or begin after shutdown has begun. Any uncommitted transaction whatsoever—even a SELECT—that involves the object or any objects that depend on it, or upon which the object itself depends, will raise this error.
- An “interesting transaction” that remains in the database through an abnormal termination by some user at some point, that involves dependencies related to this object, will raise this error.
 - ❖ Firebird considers a transaction to be “interesting” if it has been neither committed nor rolled back, since record versions will exist for which no resolution has been determined.
- You, or another suitably privileged user, previously attempted to redefine or delete this object, or another dependency-related object, and the operation was deferred because that object was in use.
 - ❖ This situation can provoke a chain of inconsistencies in your database. For example, if a gbak is performed whilst the database has objects in this state, the backup file may fail to restore

Whenever you see this error and believe that you have eliminated all possible reasons for it, take it as a sign that your database needs a validation before you proceed with any further metadata changes.

- ❖ To see a list of database procedures and their dependencies, use the **isql** command [SHOW PROCEDURE\[S\]](#). For triggers, use [SHOW TRIGGER\[S\]](#).
For information about validating a database, see [Performing a database validation](#) on page 481 in chapter 24



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Internals of the technology

When any *request* invokes a stored procedure, the current definition for that stored procedure is copied at that moment to a *metadata cache*. This copy persists for the lifetime of the request that invoked the stored procedure.

A *request* is one of the following:

- A client application that executes the stored procedure directly
- A trigger that executes the stored procedure; this includes system triggers that are part of referential integrity or check constraints
- Another stored procedure that executes the stored procedure

Altering or dropping a stored procedure takes effect immediately; new requests that invoke the altered stored procedure see the latest version. However, outstanding requests continue to see the version of the stored procedure that they first saw, even if a newer version has been created after the request's first invocation of the stored procedure.

There is no mechanism to force these outstanding requests to update their metadata cache.

A trigger or stored procedure request persists in the metadata cache whilst there are one or more clients connected to the database, regardless of whether the client makes use of the trigger or stored procedure. These requests never update as long as any client is connected to the database. These requests are emptied from the metadata cache only when the last client disconnects from the database.

❖ The only way to guarantee that all copies of a stored procedure are purged from the metadata cache is for all connections to the database to terminate. Only then are all metadata objects emptied from the metadata cache. Subsequent connections and triggers spawned by them are new requests, and they see the newest version of the stored procedure.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Stored procedures

A stored procedure is a self-contained program written in Firebird PSQL, compiled by the Firebird's internal P/L compiler and stored as executable code within the metadata of a database. Once compiled, the stored procedure can be invoked directly from an application using an EXECUTE PROCEDURE or SELECT query, according to the style of procedure which has been specified:

- procedures which perform only insert, update or delete operations are referred to as *executable stored procedures*. They are invoked with an EXECUTE PROCEDURE query.
- those which produce a multiple-row output set are referred to as selectable stored procedures. The output set is brought to the client using a SELECT query.

Stored procedures can accept *input parameters* from client applications as arguments to the invoking query. They can return a single-row set of values to applications as *output parameters*.

Firebird procedure and trigger language includes SQL data manipulation statements and some powerful extensions, including IF ... THEN ... ELSE, WHILE ... DO, FOR SELECT ... DO, system-defined exceptions and error handling.

Applications can call stored procedures using dynamic SQL statements. You can also invoke them interactively in **isql** or many of the database desktop tools recommended for use with Firebird, listed in the *Firebird Reference Guide*.

Benefits of using stored procedures

- **Modular design**

All applications that access the same database share stored procedures, thus centralizing business rules, re-using code and reducing the size of the applications

- **Streamlined maintenance**

When a procedure is modified, changes propagate automatically to all applications without the need for further recompiling on the application side, unless changes affect input or output argument sets.

- **Improved performance**

**Symbols** [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

Server-side execution of complex processing happens on the server, reducing network traffic and the overhead of operating on external sets.

- **Architectural economy**

Client applications can focus on capturing user input and managing interactive tasks while delegating complex data refinement and dependency management to a dedicated data management engine

Creating procedures

In your script or in **isql**, begin by setting the terminator symbol that will be used to mark the end of the CREATE PROCEDURE syntax. The following example will set the terminator symbol to `&`:

```
...
SET TERM & ;
```

CREATE PROCEDURE syntax

```
CREATE PROCEDURE name
[ (param datatype [, param datatype ...]) ]
[ RETURNS (param datatype [, param datatype ...]) ]
AS
<procedure_body>;
```

Header elements

In the header, declare

- the **name of the procedure**, which is required and must be unique in the database, e.g.

```
CREATE PROCEDURE MyProc
```

- any **optional input parameters** (arguments) required by the procedure, and their data types, e.g.

```
CREATE PROCEDURE MyProc
```

```
(invar1 integer, invar2 date)
```

The name of the parameter must be unique within the procedure. The data type can be any standard SQL datatype except BLOB and arrays of datatypes. The name of an input parameter need not match the name of any host parameter in the calling program.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- ❖ No more than 1,400 input parameters can be passed to a stored procedure.

- any **optional output parameters** required, and their data types, e.g.

```
CREATE PROCEDURE MyProc
  (invar 1 INTEGER, invar2 DATE)
RETURNS (outvar1 INTEGER, outvar2 VARCHAR(20), outvar3 DOUBLE PRECISION)
The name of the parameter must be unique within the procedure. The data type can be any standard SQL
datatype except BLOB and arrays of datatypes.
```

- the keyword **AS**, which is required:

```
CREATE PROCEDURE MyProc
  (invar 1 INTEGER, invar2 DATE)
RETURNS (outvar1 INTEGER, outvar2 VARCHAR(20), outvar3 DOUBLE PRECISION)
AS
```

Body elements

The syntax outline is:

```
<procedure_body> = [<variable_declaration_list>]
<block>
```

- If you have **local variables** to declare, their declarations come next. Each declaration is terminated with a colon. The syntax outline is:

```
<variable_declaration_list> =
DECLARE VARIABLE var datatype;
[DECLARE VARIABLE var datatype; ... ]
```

For example,

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

```
CREATE PROCEDURE MyProc
  (invar 1 INTEGER, invar2 DATE)
  RETURNS (outvar1 INTEGER, outvar2 VARCHAR(20), outvar3 DOUBLE PRECISION)
AS
DECLARE VARIABLE localvar integer;
DECLARE VARIABLE anothervar DOUBLE PRECISION
```

- next comes the **main code block**, starting with the keyword BEGIN and ending with the keyword END. The syntax outline is:

```
<block> =
BEGIN
<compound_statement>
[<compound_statement> ...]
END <terminator>
```

For example,

```
CREATE PROCEDURE MyProc
  (invar 1 INTEGER, invar2 DATE)
  RETURNS (outvar1 INTEGER, outvar2 VARCHAR(20), outvar3 DOUBLE PRECISION)
AS
DECLARE VARIABLE localvar integer;
DECLARE VARIABLE anothervar DOUBLE PRECISION
BEGIN
```

```
  <compound_statement>
END &
```

- The **<compound_statement>** element can be a single statement, a block of statements and embedded blocks of statements bounded by BEGIN...END pairs. Blocks can include
 - Assignment statements, to set values of local variables and input/output parameters.
 - SELECT statements, to retrieve column values. SELECT statements must have an INTO clause as the last clause; and corresponding local variable or output parameter declarations for each column selected.

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

- Control-flow statements, such as FOR SELECT ... DO, IF ... THEN, and WHILE ... DO, to perform conditional or looping tasks.
- EXECUTE PROCEDURE statements, to invoke other procedures. Recursion is allowed.
- Comments to annotate procedure code.
- Exception statements, to return error messages to applications, and WHEN statements to handle specific error conditions.
- SUSPEND and EXIT statements, that return control—and return values of output parameters—to the calling application.

Example:

```
...
BEGIN
  FOR SELECT COL1, COL2, COL3, COL4
    FROM TABLEA INTO :COL1, :COL2, :COL3 DO
      BEGIN
        <statements>
      END
    <statements>
  END &
  SET TERM ; ^

```

- ❖ Notice the termination of the entire procedure declaration with the defined terminator character, followed by a statement to reset the terminator to `;`.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

TABLE 25–3 CREATE PROCEDURE arguments

Argument	Description
<i>name</i>	Name of the procedure; must be unique among procedure, table, and view names in the database
<i>param datatype</i>	<p>Input parameters that the calling program uses to pass values to the procedure</p> <p><i>param</i>: Name of the input parameter, unique for variables in the procedure</p> <p><i>datatype</i>: A Firebird datatype</p>
<i>RETURNS param datatype</i>	<p>Output parameters that the procedure uses to return values to the calling program</p> <p><i>param</i>: Name of the output parameter, unique for variables within the procedure</p> <p><i>datatype</i>: A Firebird datatype</p> <p>The procedure returns the values of output parameters when it reaches a SUSPEND statement in the procedure body.</p>
AS	Reserved word that separates the procedure header and the procedure body
DECLARE	Declares local variables used only in the procedure
VARIABLE <i>var datatype</i>	<p>Each declaration must be preceded by DECLARE VARIABLE and followed by a semicolon (:).</p> <p><i>var</i>: Name of the local variable, unique for variables in the procedure</p>
<i>statement</i>	<p>Any single statement in Firebird procedure and trigger language</p> <p>Each statement except BEGIN and END must be followed by a semicolon (:).</p>



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Selectable stored procedures

Multiple-row output

If the procedure is designed to return a multiple-row output set to the calling application, a SUSPEND statement will be required at each iteration of the FOR loop which is used for processing the input set. For “singleton” output, the EXIT statement is sufficient to cause the single-row set to be returned.



For more information about SUSPEND, see [Altering stored procedures](#) on page 529.

SELECTs on stored procedures

In a SELECT query that retrieves values from a procedure, the column names specified must match the names and data types of the procedure’s output parameters. In an EXECUTE PROCEDURE query, the output set is returned with the output arguments as column names.

The FOR SELECT...DO construct

To retrieve multiple rows in a procedure, use the FOR SELECT ... DO construct. The syntax is:

```
FOR  
<select_expr>  
DO  
<compound_statement>;
```

FOR SELECT differs from a standard SELECT as follows:

- It is a loop construct that retrieves the row specified in the *select_expr* and performs the statement or block following DO for each row retrieved.
- The INTO clause in the *select_expr* is required and must come last.

★ This syntax allows FOR ... SELECT to use the SQL UNION clause, if needed.

For example, the following statement from a procedure selects department numbers into the local variable, RDNO, which is then used as an input parameter to the DEPT_BUDGET procedure:



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

```
FOR SELECT DEPT_NO
  FROM DEPARTMENT
 WHERE HEAD_DEPT = :DNO

INTO :RDNO DO
BEGIN
 EXECUTE PROCEDURE DEPT_BUDGET :RDNO RETURNS :SUMB;
 TOT = TOT + SUMB;
END
...;
```

Nested and recursive procedures

Nested

A stored procedure can itself execute a stored procedure. Each time a stored procedure calls another procedure, the call is said to be *nested* because it occurs in the context of a previous and still active call to the first procedure. A stored procedure called by another stored procedure is known as a *nested procedure*.

Recursive

If a procedure calls itself, it is *recursive*. Recursive procedures are useful for tasks that involve repetitive steps. Each invocation of a procedure is referred to as an *instance*, since each procedure call is a separate entity that performs as if called from an application, reserving memory and stack space as required to perform its tasks.

Stored procedures can be nested up to 1,000 levels deep. This limitation helps to prevent infinite loops that can occur when a recursive procedure provides no absolute terminating condition.

- ❖ Nested procedure calls can, nevertheless, be restricted to fewer than 1,000 levels by memory and stack limitations of the server.

The following example illustrates a recursive procedure, FACTORIAL, which calculates factorials. The procedure calls itself recursively to calculate the factorial of NUM, the input parameter.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

```
SET TERM !!!;
CREATE PROCEDURE FACTORIAL (NUM INT)
RETURNS (N_FACTORIAL DOUBLE PRECISION)
AS

DECLARE VARIABLE NUM_LESS_ONE INT;
BEGIN
IF (NUM = 1) THEN
BEGIN /***** BASE CASE: 1 FACTORIAL IS 1 ****/
N_FACTORIAL = 1;
SUSPEND;
END
ELSE
BEGIN /**** RECURSION: NUM FACTORIAL = NUM * (NUM-1) FACTORIAL ****/
NUM_LESS_ONE = NUM - 1;
EXECUTE PROCEDURE FACTORIAL NUM_LESS_ONE
RETURNING_VALUES N_FACTORIAL;
N_FACTORIAL = N_FACTORIAL * NUM;
SUSPEND;
END
END !!
SET TERM ;!!!
```

The following C code demonstrates how a host-language program would call FACTORIAL:



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

```
    . . .
printf ('\nCalculate factorial for what value? ');
scanf ('%d', &pnum);
EXEC SQL
EXECUTE PROCEDURE FACTORIAL :pnum RETURNING_VALUES :pfact;
printf ('%d factorial is %d.\n', pnum, pfact);
    . . .
```

Recursion nesting restrictions would not allow this procedure to calculate factorials for numbers greater than 1,001. Arithmetic overflow, however, occurs for much smaller numbers.

Altering stored procedures

To change a stored procedure, use ALTER PROCEDURE. This statement changes the definition of an existing stored procedure while preserving its dependencies on other objects.

It will also, in general, preserve dependencies involving other objects which depend on it. However, there is an exception to this rule, as at Firebird 1.0, which may be a bug:

- If the changes to the stored procedure alter the definitions of its input or output arguments, it will be necessary to perform a RECREATE PROCEDURE on any other stored procedure which its definition causes to be called during execution.

See [The RECREATE PROCEDURE statement](#).

Changes made to a procedure are transparent to all client applications that use the procedure—the applications do not have to be rebuilt.; you do not have to rebuild the applications. However, see [Triggers](#) on page 532 for issues of managing versions of stored procedures.

❖ Caveat: if your alterations to a stored procedure affect the type, number or and order of input and output parameters, it will be necessary to review and adjust areas of your application code that refer to these sets.



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

ALTER PROCEDURE syntax

The syntax for ALTER PROCEDURE is similar to CREATE PROCEDURE as shown in the following syntax:

```
ALTER PROCEDURE name
[ (var datatype [, var datatype ...]) ]
[RETURNS (var datatype [, var datatype ...]) ]
AS
procedure_body;
```

The procedure *name* must be the name of an existing procedure. The arguments of the ALTER PROCEDURE statement are the same as those for CREATE PROCEDURE—see [CREATE PROCEDURE arguments](#) on page 525.

The RECREATE PROCEDURE statement

RECREATE PROCEDURE is identical to CREATE PROCEDURE, except that it causes any existing procedure of the same name to be dropped, before recreating and recompiling.

- You can use it like ALTER PROCEDURE but it will not preserve existing dependencies.
- It will react to attempted dependency violations in the same ways as do alter procedure and drop procedure.

Syntax

The procedure *name* must be the name of an existing procedure.

- ❖ Take care with case-sensitivity of object names if quoted identifiers were used when creating the procedure.

The arguments of the RECREATE PROCEDURE statement are the same as those for CREATE PROCEDURE—see [CREATE PROCEDURE arguments](#) on page 525; also [CREATE PROCEDURE syntax](#).

Dropping procedures

The DROP PROCEDURE statement deletes an existing stored procedure from the database. DROP PROCEDURE can be used interactively with **isql** or in DDL script.

The following restrictions apply to dropping procedures:

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

- Only SYSDBA and the owner of a procedure can drop it.
- You cannot drop a procedure used by other procedures, triggers, or views. Alter the other metadata object so that it does not reference the procedure, commit that work, then drop the procedure.
- You cannot drop a procedure that is recursive or in a cyclical dependency with another procedure; you must alter the procedure to remove the cyclical dependency, then drop the procedure.
- You can't drop a procedure that is currently in use by an active transaction. Commit the transaction, then drop the procedure. Refer to comments about [Altering stored procedures](#).
- Only an interactive DSQL statement or a statement in a DDL script can effect a DROP PROCEDURE operation. It is not available to embedded applications.
 - ❖ If you attempt to drop a procedure and receive an error, make sure you have entered the procedure name correctly. Case-sensitivity may be an issue.

Syntax

The syntax for dropping a procedure is:

```
DROP PROCEDURE name;
```

The procedure *name* must be the name of an existing procedure.

- ❖ Take care with case-sensitivity of object names if quoted identifiers were used when creating the procedure.

The following statement deletes the ACCOUNTS_BY_CLASS procedure:

```
DROP PROCEDURE ACCOUNTS_BY_CLASS;
```



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Triggers

A *trigger* is a self-contained routine associated with a table or view that automatically performs an action when a row in the table or view is inserted, updated, or deleted.

A trigger is never called directly. Instead, when an application or user attempts to INSERT, UPDATE, or DELETE a row in a table, any triggers associated with that table and operation are automatically executed, or *fired*.

Triggers can make use of exceptions and can trigger events. They can also call stored procedures.

Benefits of using triggers

- **Automatic enforcement of data restrictions**, to make sure users enter only valid values into columns.
- **Reduced application maintenance**, since changes to a trigger are automatically reflected in all applications that use the associated table without the need to recompile and relink.
- **Automatic logging of changes to tables**. An application can keep a running log of changes with a trigger that fires whenever a table is modified.
- **Automatic notification of changes to the database** with event alerters in triggers.

There are two ways to create, alter, and drop triggers with **isql**:

- Interactively
- With a DDL script

It is preferable to use scripts, because it is easier to modify these files and provide a record of the changes made to the database.

Creating triggers

A trigger is defined with the CREATE TRIGGER statement, which is composed of a *header* and a *body*. The trigger header contains:

- A trigger name, unique within the database.
- A table name, identifying the table with which to associate the trigger.
- Statements that determine when the trigger fires.

The trigger body contains:

**Symbols** [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

- An optional list of local variables and their datatypes.
- A block of statements in Firebird procedure and trigger language, bracketed by BEGIN and END. These statements are performed when the trigger fires. A block can itself include other blocks, so that there may be many levels of nesting.

CREATE TRIGGER syntax

The syntax of CREATE TRIGGER is:

```
CREATE TRIGGER name FOR {table | view}  
[ACTIVE | INACTIVE]  
{BEFORE | AFTER} {DELETE | INSERT | UPDATE}  
[POSITION number]  
AS <trigger_body>  
<trigger_body> = [<variable_declaration_list>] <block>  
<variable_declaration_list> =DECLARE VARIABLE variable datatype;  
[DECLARE VARIABLE variable datatype; ...]  
<block> =  
BEGIN  
<compound_statement> [<compound_statement> ...]  
END  
<compound_statement> = <block> | statement;
```

TABLE 25–4 CREATE TRIGGER arguments

Argument	Description
<i>name</i>	Name of the trigger. The name must be unique in the database.
<i>table</i>	Name of the table or view that causes the trigger to fire when the specified operation occurs on the table or view.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

TABLE 25–4 CREATE TRIGGER arguments (*continued*)

Argument	Description
ACTIVE INACTIVE	Optional. Specifies trigger action at transaction end: ACTIVE: (Default). Trigger takes effect. INACTIVE: Trigger does not take effect.
BEFORE AFTER	Required. Specifies whether the trigger fires: BEFORE: Before associated operation. AFTER: After associated operation. Associated operations are DELETE, INSERT, or UPDATE.
DELETE INSERT UPDATE	Specifies the table operation that causes the trigger to fire.
POSITION <i>number</i>	Specifies firing order for triggers before the same action or after the same action. <i>number</i> must be an integer between 0 and 32,767, inclusive. Lower-number triggers fire first. Default: 0 = first trigger to fire. Triggers for a table need not be consecutive. Triggers on the same action with the same position number will fire in alphabetic order by name.

Trigger header elements

Everything before the AS clause in the CREATE TRIGGER statement forms the trigger header. The header must specify the name of the trigger and the name of the associated table or view. The table or view must exist before it can be referenced in CREATE TRIGGER.

The trigger name must be unique among triggers in the database. Using the name of an existing trigger or a system-supplied constraint name results in an error.

The remaining clauses in the trigger header determine when and how the trigger fires:

- The *trigger status*, ACTIVE or INACTIVE, determines whether a trigger is activated when the specified operation occurs. ACTIVE is the default, meaning the trigger fires when the operation occurs. Setting

**Symbols** [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

status to INACTIVE with ALTER TRIGGER is useful when developing and testing applications and triggers.

- The *trigger phase indicator*, BEFORE or AFTER, determines when the trigger fires relative to the specified operation. BEFORE specifies that trigger actions are performed before the operation. AFTER specifies that trigger actions are performed after the operation.
- The *trigger statement indicator* specifies the SQL operation that causes the trigger to fire: INSERT, UPDATE, or DELETE. Exactly one indicator must be specified. To use the same trigger for more than one operation, duplicate the trigger with another name and specify a different operation.
- The optional *sequence indicator*, POSITION *number*, specifies the order in which the trigger fires in relation to other triggers on the same table and event. *number* can be any integer between zero and 32,767. The default is zero. Lower-numbered triggers fire first. Multiple triggers can have the same position number; they will fire in random order.

The following example demonstrates how the POSITION clause determines trigger firing order. Here are four headers of triggers for the ACCOUNTS table:

```
CREATE TRIGGER A FOR ACCOUNTS BEFORE UPDATE POSITION 5 AS ...
CREATE TRIGGER B FOR ACCOUNTS BEFORE UPDATE POSITION 0 AS ...
CREATE TRIGGER C FOR ACCOUNTS AFTER UPDATE POSITION 5 AS ...
CREATE TRIGGER D FOR ACCOUNTS AFTER UPDATE POSITION 3 AS ...
```

When this update takes place:

```
UPDATE ACCOUNTS SET C = 'canceled' WHERE C2 = 5;
```

The following sequence of events happens: trigger B fires, A fires, the update occurs, trigger D fires, then C fires.

The trigger body

Everything following the AS reserved word in the CREATE TRIGGER statement forms the procedure body. In all Firebird code modules, the body consists of an optional list of local variable declarations followed by a block of statements. Programming a trigger body is exactly the same as programming a stored procedure body. For details, refer to [Coding the body of the code module](#) and [Body elements](#).

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

Triggers can include EXECUTE PROCEDURE statements to invoke stored procedures. For the syntax of execute procedure, see [Using executable procedures](#) on page 539. Full syntax can be found in the Statement and Function Reference of the *Firebird Reference Guide*.

Trigger-specific extension

A special Firebird SQL code extension specific to trigger language is NEW and OLD context variables, which store previous (old) column values and refer to the value to be inserted into columns in new rows or rows being updated. For more details, see [NEW and OLD context variables](#) on page 504.

Altering triggers

To update a trigger definition, use ALTER TRIGGER. A trigger can be altered only by its creator.

ALTER TRIGGER may change:

- Only trigger header information, including the trigger activation status, when it performs its actions, the event that fires the trigger, and the order in which the trigger fires compared to other triggers.
- Only trigger body information, the trigger statements that follow the AS clause.
- Both trigger header and trigger body information. In this case, the new trigger definition replaces the old trigger definition.

System-defined triggers

To alter a trigger defined automatically by a CHECK or other constraint on a table, you must use ALTER TABLE to change the table definition. For more information, see [Using ALTER TABLE](#) on p. 343 of chapter 17.

Syntax

The ALTER TRIGGER syntax is as follows:

```
ALTER TRIGGER name
[ACTIVE | INACTIVE]
[ {BEFORE | AFTER} {DELETE | INSERT | UPDATE} ]
[POSITION number]
AS <trigger_body>;
```

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

The syntax of ALTER TRIGGER is the same as CREATE TRIGGER, except:

- The CREATE reserved word is replaced by ALTER.
- FOR *table* is omitted. ALTER TRIGGER cannot be used to change the table with which the trigger is associated.
- The statement need only include parameters that are to be altered in the existing trigger, with certain exceptions listed in the following sections.

Altering a trigger header

When used to change only a trigger header, ALTER TRIGGER requires at least one altered setting after the trigger name. Any setting omitted from ALTER TRIGGER remains unchanged.

The following statement makes the trigger, SAVE_SALARY_CHANGE, inactive:

```
ALTER TRIGGER SAVE_SALARY_CHANGE INACTIVE;
```

If the *phase indicator* (BEFORE or AFTER) is altered, then the operation (UPDATE, INSERT, or DELETE) must also be specified. For example, the following statement reactivates the trigger, VERIFY_FUNDS, and specifies that it fire before an UPDATE instead of after:

```
ALTER TRIGGER SAVE_SALARY_CHANGE  
ACTIVE  
BEFORE UPDATE;
```

Altering a trigger body

When a trigger body is altered, the new body definition replaces the old definition. When used to change only a trigger body, ALTER TRIGGER need not contain any header information other than the trigger's name.

To make changes to a trigger body:

- 1 Copy the original data definition file used to create the trigger. Alternatively, use **isql -extract** to extract a trigger from the database to a file.
- 2 Edit the file, changing CREATE to ALTER, and delete all trigger header information after the trigger name and before the AS reserved word.
- 3 Change the trigger definition as desired. Retain whatever is still useful. The trigger body must remain syntactically and semantically complete.

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

For example, the following ALTER statement modifies the previously introduced trigger, SET_CUST_NO, to insert a row into the (assumed to be previously defined) table, NEW_CUSTOMERS, for each new customer.

```
SET TERM !! ;
ALTER TRIGGER SET_CUST_NO
BEFORE INSERT AS
BEGIN
new.cust_no = GEN_ID(CUST_NO_GEN, 1);
INSERT INTO NEW_CUSTOMERS(new.cust_no, 'TODAY')
END !!
SET TERM ; !!
```

Dropping triggers

During database design and application development, a trigger may cease to be useful. To remove a trigger permanently, use DROP TRIGGER.

The following restrictions apply to dropping triggers:

- Only the creator of a trigger can drop it.
 - Triggers currently in use cannot be dropped.
- ★ To remove a trigger temporarily, use ALTER TRIGGER and specify INACTIVE in the header.
- ❖ You cannot drop a trigger if it is in use by a CHECK constraint (a system-defined trigger). Use ALTER TABLE to remove or modify the CHECK clause that defines the trigger.

Syntax

The DROP TRIGGER syntax is as follows:

```
DROP TRIGGER name;
```

The trigger *name* must be the name of an existing trigger. The following example drops the trigger, SET_CUST_NO:

```
DROP TRIGGER SET_CUST_NO;
```



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Implementing stored procedures and triggers

Uses for stored procedures

Stored procedures can be used in applications in a variety of ways.

- Selectable procedures are used in place of a table or view in a SELECT statement.
- Executable procedures are used with an EXECUTE PROCEDURE statement to perform a single operation or set of operations on the server side.
- A stored procedure can be invoked by another stored procedure or a trigger. It can call itself recursively.

All stored procedures are defined with CREATE PROCEDURE and have the same syntax. Differences of usage depend on how the procedure is written. The abiding rule is to design each procedure *atomically*—to do a single task or set of tasks within the predictable context of a single transaction, to be committed as a unit if everything succeeds or rolled back as a unit if something fails.

Using executable procedures

An executable procedure is invoked with EXECUTE PROCEDURE. It can return at most one row. To execute a stored procedure in **isql**, use the following syntax:

```
EXECUTE PROCEDURE name [ ( ] [ param [, param ...] ] [ ] );
```

The procedure *name* must be specified, and each *param* is an input parameter value (a constant). All input parameters required by the procedure must be supplied.

❖ In **isql**, do not supply *output parameters* nor use RETURNING_VALUES in the EXECUTE PROCEDURE statement, even if the procedure returns values. **isql** automatically displays output parameters.

To execute the procedure, DEPT_BUDGET, from **isql**, enter:



Symbols [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

EXECUTE PROCEDURE DEPT_BUDGET 110;

isql displays this output:

TOT

=====

1700000.00

Using selectable procedures

A select procedure is used in place of a table or view in a SELECT statement and can return a single row or multiple rows.

The advantages of select procedures over tables or views are:

- They can take input parameters that can affect the output.
- They can contain logic not available in normal queries or views.
- They can return rows from multiple tables using UNION.

Syntax for SELECT from a procedure

```
SELECT <col_list> from name ([param [, param ...]])  
WHERE <search_condition>  
ORDER BY <order_list>;
```

- the procedure *name* must be specified
- in isql each *param* is a constant passed to the corresponding input parameter. All input parameters required by the procedure must be supplied.
- the *col_list* is a comma-delimited list of output parameters returned by the procedure, or * to select all columns.
- a WHERE clause can specify a *search_condition* that limits the output set to a subset of rows.
- an ORDER BY clause can specify a sort order for the rows returned.



For more information on SELECT, see the appropriate entry in the Statement and Function Reference of the *Firebird Reference Guide*. For relevant related topics, see [Queries](#) on p. 116 of chapter 9.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Examples

The following code defines the procedure, GET_EMP_PROJ, which returns EMP_PROJ, the project numbers assigned to an employee, when it is passed the employee number, EMP_NO, as the input parameter.

```
SET TERM !! ;
CREATE PROCEDURE GET_EMP_PROJ (EMP_NO SMALLINT)
RETURNS (EMP_PROJ SMALLINT) AS
BEGIN
FOR SELECT PROJ_ID
FROM EMPLOYEE_PROJECT
WHERE EMP_NO = :EMP_NO
INTO :EMP_PROJ
DO
SUSPEND ;
END !!
```

The following statement selects from GET_EMP_PROJ:

```
SELECT * FROM GET_EMP_PROJ(24);
```

The output is:

```
PROJ_ID
=====
DGPII
GUIDE
```

The following select procedure, ORG_CHART, is designed to display an organizational chart:

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

```
CREATE PROCEDURE ORG_CHART
RETURNS (HEAD_DEPT CHAR(25), DEPARTMENT CHAR(25),
MNGR_NAME CHAR(20), TITLE CHAR(5), EMP_CNT INTEGER)
AS
DECLARE VARIABLE MNGR_NO INTEGER;
DECLARE VARIABLE DNO CHAR(3);
BEGIN
FOR SELECT H.DEPARTMENT, D.DEPARTMENT, D.MNGR_NO, D.DEPT_NO
FROM DEPARTMENT D
LEFT OUTER JOIN DEPARTMENT H ON D.HEAD_DEPT = H.DEPT_NO
ORDER BY D.DEPT_NO
INTO :HEAD_DEPT, :DEPARTMENT, :MNGR_NO, :DNO
DO
BEGIN
IF (:MNGR_NO IS NULL) THEN
BEGIN
MNGR_NAME = '--TBH--';
TITLE = '';
END
ELSE
SELECT FULL_NAME, JOB_CODE
FROM EMPLOYEE
WHERE EMP_NO = :MNGR_NO
INTO :MNGR_NAME, :TITLE;
SELECT COUNT(EMP_NO)
FROM EMPLOYEE
WHERE DEPT_NO = :DNO
INTO :EMP_CNT;
SUSPEND;
```



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

END

END !!

ORG_CHART is invoked from **isql** as follows:

```
SELECT * FROM ORG_CHART;
```

For each department, the procedure displays the department name, the department's "head department" (managing department), the department manager's name and title, and the number of employees in the department.

HEAD_DEPT	DEPARTMENT	MNGR_NAME	TITLE	EMP_CNT
=====	=====	=====	====	=====
Corporate Headquarters		Bender, Oliver H.	CEO	2
Corporate Headquarters	Sales and Marketing	MacDonald, Mary S.	VP	2
Sales and Marketing	Pacific Rim Headquarters	Baldwin, Janet	Sales	2
Pacific Rim Headquarters	Field Office: Japan	Yamamoto, Takashi	SRep	2
Pacific Rim Headquarters	Field Office: Singapore	-TBH-		0

❖ ORG_CHART must be invoked as a selectable procedure to display the full organization. If called with EXECUTE PROCEDURE, the procedure would terminate the first time it encountered the SUSPEND statement, returning only the set of data for Corporate Headquarters.

SELECT can specify columns to retrieve from a procedure. For example, if ORG_CHART is invoked as follows:

```
SELECT DEPARTMENT FROM ORG_CHART;
```

then only the second column, DEPARTMENT, is displayed.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Using WHERE and ORDER BY clauses

A SELECT from a stored procedure can contain WHERE and ORDER BY clauses, just as in a SELECT from a table or view.

- a WHERE clause limits the results returned by the procedure to rows matching the search condition. For example, the following statement returns only those rows where the HEAD_DEPT is Sales and Marketing:

```
SELECT * FROM ORG_CHART WHERE HEAD_DEPT = 'Sales and Marketing';
```

The stored procedure then returns only the matching rows, for example

HEAD_DEPT	DEPARTMENT	Mngr_Name	Title	EMP_CNT
=	=	=	====	=====
Sales and Marketing	Pacific Rim Headquarters	Baldwin, Janet	Sales	2
Sales and Marketing	European Headquarters	Reeves, Roger	Sales	3
Sales and Marketing	Field Office: East Cost	Weston, K. J.	SRep	2

- an ORDER BY clause can be used to sort the results returned by the procedure. For example, the following statement orders the results by EMP_CNT, the number of employees in each department, in ascending order (the default):

```
SELECT * FROM ORG_CHART ORDER BY EMP_CNT;
```

Selecting aggregates from procedures

In addition to selecting values from a procedure, you can use aggregate functions. For example, to use ORG_CHART to display a count of the number of departments, use the following statement:

```
SELECT COUNT(DEPARTMENT) FROM ORG_CHART;
```

The results are:



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

COUNT

=====

24

Similarly, to use ORG_CHART to display the maximum and average number of employees in each department, use the following statement:

```
SELECT MAX(EMP_CNT), AVG(EMP_CNT) FROM ORG_CHART;
```

The results are:

MAX AVG

===== =====

5 2

- ❖ If a procedure encounters an error or exception, the aggregate functions do not return the correct values, since the procedure terminates before all rows are processed.

Procedures for combined use

Although it is possible, for example, to design a selectable procedure that executes a DML operation in the course of constructing an output set, in most circumstances it is not recommended. A selectable stored procedure is designed to output a set of data to the client, in the transaction context which invokes it. Until the client application has finished using that output set, the transaction remains uncommitted. If DML operations are included in the code that generates the output set, those DML requests remain uncommitted until the transaction is completed by the client. In particular, data have the potential to be stored inconsistently if values from the stored procedure output are passed as parameters to operations in other transactions.

Viewing arrays with stored procedures

If a table contains columns defined as arrays, you cannot view the data in the column with a simple SELECT statement, since only the array ID is stored in the table. Arrays can be used to display array values, as long as the dimensions and datatype of the array column are known in advance.

For example, in the **employee** database, the JOB table has a column named LANGUAGE_REQ containing the languages required for the position. The column is defined as an array of five VARCHAR(15).

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

In **isql**, if you perform a simple SELECT statement, such as:

```
SELECT JOB_CODE, JOB_GRADE, JOB_COUNTRY, LANGUAGE_REQ FROM JOB;  
part of the results look like this:
```

JOB_CODE	JOB_GRADE	JOB_COUNTRY	LANGUAGE_REQ
=====	=====	=====	=====
.	.	.	.
Sales	3	USA	<null>
Sales	3	England	20:af
SRep	4	USA	20:b0
SRep	4	England	20:b2
SRep	4	Canada	20:b4

The following code is an example of how to use a stored procedure to view the contents of the LANGUAGE_REQ column. The procedure uses a FOR...SELECT loop to retrieve each row from JOB for which LANGUAGE_REQ is not NULL. Then a WHILE loop retrieves each element of the LANGUAGE_REQ array and returns the value to the calling application (in this case, **isql**).

```
SET TERM !! ;  
CREATE PROCEDURE VIEW_LANGS  
RETURNS (code VARCHAR(5), grade SMALLINT, cty VARCHAR(15),  
lang VARCHAR(15))  
AS  
DECLARE VARIABLE i INTEGER;  
BEGIN  
FOR SELECT JOB_CODE, JOB_GRADE, JOB_COUNTRY  
FROM JOB  
WHERE LANGUAGE_REQ IS NOT NULL
```



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

```
INTO :code, :grade, :cty
DO
BEGIN
i = 1;
WHILE (i <= 5) DO
BEGIN
SELECT LANGUAGE_REQ[:i] FROM JOB
WHERE ((JOB_CODE = :code) AND (JOB_GRADE = :grade)
AND (JOB_COUNTRY = :cty)) INTO :lang;
i = i + 1;
SUSPEND;
END
END
END! !
SET TERM ; ! !
```

For example, the output if this procedure is invoked with:

```
SELECT * FROM VIEW_LANGS;
```

is

CODE	GRADE	CTY	LANG
=====	=====	=====	=====
Eng	3	Japan	Japanese
Eng	3	Japan	Mandarin
Eng	3	Japan	English
Eng	3	Japan	
Eng	3	Japan	
Eng	4	England	English
Eng	4	England	German



Symbols [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

Eng 4 England French

...

This procedure can be easily modified to return only the language requirements for a particular job, when passed JOB_CODE, JOB_GRADE, and JOB_COUNTRY as input parameters.

Uses for triggers

Triggers are a powerful feature with a variety of uses. Among the ways that triggers can be used are:

- To make correlated updates. For example, to keep a log file of changes to a database or table.
- To enforce data restrictions, so that only valid data is entered in tables.
- Automatic transformation of data. For example, to automatically convert text input to uppercase.
- To notify applications of changes in the database using event alerters.
- To perform cascading referential integrity updates.
- To make a read-only view updatable. For details, see [Making read-only views updatable](#) on p. 370 of chapter 19.

Triggers are stored as objects in a database, like stored procedures and exceptions. Once defined to be ACTIVE, they remain active until deactivated with ALTER TRIGGER or removed from the database with DROP TRIGGER. A trigger is never explicitly called—an active trigger fires automatically when the specified DML operation occurs on the specified table.

- ❖ If a trigger performs an action that causes it to fire again—or fires another trigger that performs an action that causes it to fire—an infinite loop results. For this reason, it is important to ensure that a trigger's actions never cause the trigger to fire, even indirectly. For example, an endless loop will occur if a trigger fires on INSERT to a table and then performs an INSERT into the same table.

Triggers and transactions

Triggers operate within the context of the transaction in the program where they are fired. Triggers are considered part of the calling program's current unit of work.

If triggers are fired in a transaction, and the transaction is rolled back, then any actions performed by the triggers are also rolled back.



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

Security

Procedures and triggers can be granted privileges on tables, just as users or roles can be granted privileges. Use the GRANT statement, but instead of using TO *username*, use TO TRIGGER *trigger_name* or TO PROCEDURE *procedure_name*. Privileges for procedures and triggers can be revoked similarly using REVOKE.

For more information about GRANT and REVOKE, see [Database-level security](#) on p. 429 of chapter 22.

When a user performs an action that fires a trigger, or invokes a stored procedure, the code module will have privileges to perform its actions if one of the following conditions is true:

- The code module has privileges for the action.
- The user has privileges for the action.

So, for example, if a user performs an UPDATE of table A, which fires a trigger, and the trigger performs an INSERT on table B, the INSERT will occur if the user has INSERT privileges on the table or the trigger has insert privileges on the table.

If there are insufficient privileges for a trigger or procedure to perform its actions, Firebird sets the appropriate SQLCODE error number. The code module can handle this error with a WHEN clause. If it does not handle the error, an error message will be returned to the application, and the actions of the code module will be undone. If the code module is a trigger, the DML operation in which the error occurred will be undone also.

For more details, refer to [Handling exceptions](#).

Error trapping and handling

An *exception* is a named error message that can be raised in a stored procedure and returned to the calling application in the error status array. Exceptions are objects stored in a database, where they can be used by any procedure that needs them.

A stored procedure or trigger can watch for an error condition and raise the named exception with EXCEPTION *name*. When raised, the exception can be *handled* in the procedure by a WHEN EXCEPTION *name* DO... block. If left unhandled, it will cause execution to terminate and return its associated error message text to the calling program.

An exception must be created and committed before it can be raised.



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

Creating exceptions

To create an exception, use the following CREATE EXCEPTION syntax:

```
CREATE EXCEPTION name '<message>' ;
```

For example, the following statement creates an exception named REASSIGN_SALES:

```
CREATE EXCEPTION REASSIGN_SALES 'Reassign the sales records  
before deleting this employee.' ;
```

Altering exceptions

To change the message returned by an exception, use the following syntax:

```
ALTER EXCEPTION name '<message>' ;
```

Only the creator of an exception can alter it. For example, the following statement changes the text of the exception created in the previous section:

```
ALTER EXCEPTION REASSIGN_SALES 'Can't delete employee--Reassign  
Sales' ;
```

Considerations

- You can alter an exception even when a database object depends on it.
- If the exception is raised by a trigger, you cannot drop the exception unless you first drop the trigger or stored procedure. Use ALTER EXCEPTION instead.

Dropping exceptions

To delete an exception, use the following syntax:

```
DROP EXCEPTION name ;
```

For example, the following statement drops the exception, REASSIGN_SALES:

```
DROP EXCEPTION REASSIGN_SALES ;
```



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Restrictions

- Only the creator of an exception can drop it.
- Exceptions used in existing procedures and triggers cannot be dropped.
- Exceptions currently in use cannot be dropped.

★ In **isql**, SHOW PROCEDURE[S] displays a list of *dependencies*, the procedures, exceptions, and tables which the stored procedure uses. SHOW PROCEDURE *name* displays the body and header information for the named procedure.

SHOW TRIGGER[S] *table* displays the triggers defined for *table*. SHOW TRIGGER *name* displays the body and header information for the named trigger.

Raising an exception code

To raise an exception in a stored procedure or trigger, use the following syntax:

```
EXCEPTION name;
```

where *name* is the name of an exception that already exists in the database.

When an exception is raised and *not handled* by the procedure or trigger, it does the following:

- Terminates the procedure in which it was raised and undoes any actions performed (directly or indirectly) by the procedure.
- Returns an error message to the calling application. In **isql**, the error message is displayed on the screen.

The following statements raise the exception, REASSIGN_SALES:

```
IF (any_sales > 0) THEN  
EXCEPTION REASSIGN_SALES;
```

If an exception is *handled* with a WHEN...DO statement, it behaves differently. Exception handling is discussed in the next section.

★ An exception that is handled does *not* return an error message.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Handling exceptions

Instead of terminating when an exception occurs, a procedure or trigger can respond to and perhaps correct the error condition by handling the exception. When an exception is raised, the code module does the following:

- seeks a WHEN statement that handles the exception. If one is not found, it terminates execution of the BEGIN ... END block containing the exception and undoes any actions performed in the block.
- backs out one level to the surrounding BEGIN ... END block and continues to seek downward for a WHEN statement that handles the exception. It continues to back out level-by-level, until one is found. If no WHEN statement is found, the procedure terminates and undoes all its actions.

When control jumps into a WHEN statement, the program module

- performs the ensuing statement or block of statements specified by the WHEN statement that handles the exception.
- returns program control to the block in the procedure following the WHEN statement.

Procedures and triggers can include a WHEN...DO statement to handle three kinds of exception:

- exceptions raised by EXCEPTION statements in the current procedure, in a nested procedure, or in a trigger fired as a result of actions by such a procedure.
- standard SQL errors reported in the context variable SQLCODE.
- Firebird status array error codes for finer-grained database error conditions, reported in the context variable GDSCODE.

Using the ANY keyword in place of any of these explicit exception types will cause any exception to be handled by the code in the WHEN...DO block.

- An exception that is handled does *not* return an error message.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Syntax for a WHEN...DO statement

```
WHEN {<error> [, <error> ...] | ANY}
DO <compound_statement>
<error> =
{EXCEPTION exception_name | SQLCODE number | GDSCODE errcode}
```

WHEN must be the last statement in a BEGIN ... END block. It should come after SUSPEND, if present.



For a complete list of Firebird status array error codes and SQLCODE values, see *Firebird Reference Guide*—[Error Codes and Messages](#) (ch. 8 p. 291).

SQL exceptions

After each SQL statement executes, the context variable SQLCODE contains a status code indicating the success or failure of the statement. SQLCODE can also contain a warning status—a positive number lower than 100—indicating a piece of status information indicating something to the procedure that is not an exception condition. For example, a warning status is returned when there are no more rows to retrieve in a FOR SELECT loop.

Example

If a code module attempts to insert a duplicate value into a column defined as a PRIMARY KEY, Firebird returns SQLCODE -803. This error can be handled in a procedure with the following statement:

```
WHEN SQLCODE -803
DO
BEGIN
...<statements to handle a primary key violation>...
END
...
```

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

The following procedure includes a WHEN statement to handle SQLCODE -803 (attempt to insert a duplicate value in a UNIQUE key column). If the first column in TABLE1 is a UNIQUE key, and the value of parameter A is the same as one already in the table, then SQLCODE -803 is generated, and the WHEN statement sets an error message returned by the procedure.

```
SET TERM !!!;
CREATE PROCEDURE NUMBERPROC (A INTEGER, B INTEGER)
RETURNS (E CHAR(60)) AS
BEGIN
BEGIN
INSERT INTO TABLE1 VALUES (:A, :B);
WHEN SQLCODE -803 DO
E = 'Error Attempting to Insert in TABLE1 - Duplicate Value.';
END;
END !!
SET TERM; !!!
```

Firebird-specific errors

Procedures can also handle the Firebird-specific errors—those that would be passed back as 9-digit codes in error status array if exceptions go unhandled.

For example, suppose a statement in a procedure attempts to update a row already updated by another transaction, but not yet committed. In this case, the procedure might receive a Firebird error LOCK_CONFLICT. The procedure could handle this condition by retrying its update, in the expectation that the other transaction might have rolled back its changes and released its locks.

To handle Firebird error codes, use the following syntax:

```
WHEN GDSCODE errcode DO <compound_statement>;
```



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Exception-handling considerations

When a procedure encounters an exception—be it a SQLCODE error, GDSCODE error, or user-defined exception—in a *selectable procedure*, the statements since the last SUSPEND are undone. Any rows already output from previous calls to SUSPEND are unaffected and remain available to the client.

- ❖ SUSPEND should *not* be used in executable procedures. To ensure that the entire procedure gets undone when an exception occurs, these procedures should be terminated using only the EXIT statement.

Example 1

In the following example, a simple executable procedure attempts to insert the same values twice into the PROJECT table.

```
SET TERM !! ;
CREATE PROCEDURE NEW_PROJECT
(id CHAR(5), name VARCHAR(20), product VARCHAR(12))
RETURNS (result VARCHAR(80))
AS
BEGIN
INSERT INTO PROJECT (PROJ_ID, PROJ_NAME, PRODUCT)
VALUES (:id, :name, :product);
result = 'Values inserted OK.';
INSERT INTO PROJECT (PROJ_ID, PROJ_NAME, PRODUCT)
VALUES (:id, :name, :product);
result = 'Values Inserted Again.';
EXIT;
WHEN SQLCODE -803 DO
BEGIN
result = 'Could Not Insert Into Table - Duplicate Value';
```



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

```
EXIT;  
END  
END!!  
SET TERM ; !!
```

Invoke the procedure with a statement such as:

```
EXECUTE PROCEDURE NEW_PROJECT 'XXX', 'Project X', 'N/A';
```

The second INSERT generates an error (SQLCODE -803, "invalid insert—no two rows can have duplicate values."). The procedure returns the string, "Could Not Insert Into Table - Duplicate Value," as specified in the WHEN clause, and the entire procedure is undone.

Example 2

The next example is written as a select procedure, and invoked with the SELECT statement that follows it:

```
...  
INSERT INTO PROJECT (PROJ_ID, PROJ_NAME, PRODUCT)  
VALUES (:id, :name, :product);  
result = 'Values inserted OK.';  
SUSPEND;  
INSERT INTO PROJECT (PROJ_ID, PROJ_NAME, PRODUCT)  
VALUES (:id, :name, :product);  
result = 'Values Inserted Again.';  
SUSPEND;  
WHEN SQLCODE -803 DO  
  
BEGIN  
result = 'Could Not Insert Into Table - Duplicate Value';  
EXIT;  
END
```

This statement invokes it:

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

```
SELECT * FROM SIMPLE('XXX', 'Project X', 'N/A');
```

The first INSERT is performed, and SUSPEND returns the result string, "Values Inserted OK." The second INSERT generates the error because no statements have been performed since the last SUSPEND, and no statements are undone. The WHEN statement appends, "Could Not Insert Into Table - Duplicate Value" to the previous result string and returns it to the client in the error status array.

The select procedure successfully performs the insert, while the executable procedure does not.

Example 3

The next example is a more complex stored procedure that demonstrates SQLCODE error handling and exception handling. It is based on the previous example of a select procedure, and does the following:

- Accepts a project ID, name, and product type, and ensures that the ID is in all capitals, and the product type is acceptable.
- Inserts the new project data into the PROJECT table, and returns a string confirming the operation, or an error message saying the project is a duplicate.
- Uses a FOR ... SELECT loop with a correlated subquery to get the first three employees not assigned to any project and assign them to the new project using the ADD_EMP_PROJ procedure.
- If the CEO's employee number is selected, raises the exception, CEO, which is handled with a WHEN statement that assigns the CEO's administrative assistant (employee number 28) instead to the new project.

Note that the exception, CEO, is handled within the FOR ... SELECT loop, so that only the block containing the exception is undone, and the loop and procedure continue after the exception is raised.

```
CREATE EXCEPTION CEO 'Can't Assign CEO to Project.';  
SET TERM !! ;  
CREATE PROCEDURE NEW_PROJECT  
(id CHAR(5), name VARCHAR(20), product VARCHAR(12))  
RETURNS (result VARCHAR(30), num smallint)  
AS  
DECLARE VARIABLE emp_wo_proj smallint;  
DECLARE VARIABLE i smallint;
```



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

```
BEGIN
  id = UPPER(id); /* Project id must be in uppercase. */
  INSERT INTO PROJECT (PROJ_ID, PROJ_NAME, PRODUCT)
    VALUES (:id, :name, :product);
  result = 'New Project Inserted OK.';
  SUSPEND;
/* Add Employees to the new project */
  i = 0;
  result = 'Project Got Employee Number:';
  FOR SELECT EMP_NO FROM EMPLOYEE
    WHERE EMP_NO NOT IN (SELECT EMP_NO FROM EMPLOYEE_PROJECT)
    INTO :emp_wo_proj
  DO
    BEGIN
      IF (i < 3) THEN
        BEGIN
          IF (emp_wo_proj = 5) THEN
            EXCEPTION CEO;
          EXECUTE PROCEDURE ADD_EMP_PROJ :emp_wo_proj, :id;
          num = emp_wo_proj;
        SUSPEND;
      END
      ELSE
        EXIT;
      i = i + 1;
    WHEN EXCEPTION CEO DO
      BEGIN
        EXECUTE PROCEDURE ADD_EMP_PROJ 28, :id;
        num = 28;
      END
    END
  END
```



Symbols [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

```
SUSPEND;  
END  
END  
/* Error Handling */  
WHEN SQLCODE -625 DO  
BEGIN  
IF ((:product <> 'software') OR (:product <> 'hardware') OR  
(:product <> 'other') OR (:product <> 'N/A')) THEN  
result = 'Enter product: software, hardware, other, or N/A';  
END  
WHEN SQLCODE -803 DO  
result = 'Could not insert into table - Duplicate Value';  
END!!  
SET TERM ; !!
```

Calling this procedure in isql::

```
SELECT * FROM NEW_PROJECT('XYZ', 'Alpha project', 'software');
```

results in:

```
RESULT NUM  
===== =====  
New Project Inserted OK. <null>  
Project Got Employee Number: 28  
Project Got Employee Number: 29  
Project Got Employee Number: 36
```

Example 4—exception handled in a trigger

A trigger that fires when the EMPLOYEE table is updated might compare the employee's old salary and new salary, and raise an exception if the salary increase exceeds 50%. The exception could return this message:

```
New salary exceeds old by more than 50%. Cannot update record.
```



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Handling events on a client

Firebird events provide a signaling mechanism by which stored procedures and triggers can pass an alert to client applications when other applications commit changes to data. The client applications are set up to "listen" for specific events through the server-client interface, avoiding the CPU and bandwidth cost of polling for changes.

When a transaction commits, any events that occurred can be posted to listening client applications, which can then respond to the event by, for example, refreshing open datasets upon being alerted to changes.

 For details of how to post events from stored procedures and triggers, refer to [Event alerters](#) in this chapter.

How events work

A Firebird event is a signal passed by a trigger or a stored procedure to the InterBase *event manager* to announce that a specific action occurred—usually a database change such as an INSERT, UPDATE, or DELETE. Events are passed by triggers or stored procedures only when the transaction under which they occur is committed.

The event mechanism consists of three parts:

- An application executing a trigger or stored procedure that posts an event to the *event manager*.
- The event manager that maintains an event queue and notifies applications when an event occurs.
- An application that registers interest in the event and waits for it to occur.

The signaler

A trigger or stored procedure must signal the occurrence of an event, usually a database change such as an INSERT, UPDATE, or DELETE, by using the POST_EVENT statement. POST_EVENT alerts the event manager to the occurrence of an event after a transaction is committed. POST_EVENT is a stored procedure and trigger language extension, available only within stored procedures and triggers.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

The event manager

The event manager records events posted to it by triggers and stored procedures and maintains a list of applications that have registered an interest in events. The event manager responds to a new event by notifying interested applications.

Responders

Listening applications can interact with specific events that might be posted by a trigger or stored procedure by:

- 1 Indicating an interest in the events to the event manager.
- 2 Waiting for event notification.
- 3 Determining which event occurred (if an application is waiting for more than one event to occur).

Event alerters

A trigger or stored procedure that posts an event is sometimes called an event alerter. For example, the following `isql` script creates a trigger that posts an event to the event manager whenever any application inserts data in a table:

```
SET TERM !! ;
CREATE TRIGGER POST_NEW_ORDER FOR SALES
ACTIVE
AFTER INSERT
POSITION 0
AS
BEGIN
POST_EVENT 'new_order';
END
!!
SET TERM ; !!
```

❖ Event names are restricted to 15 characters.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Registering interest in events

An application must inform the event manager that it wants to be notified about a particular event before it can begin listening for the event. To register interest in an event, use the EVENT INIT statement. EVENT INIT requires two arguments:

- An application-specific request handle to pass to the event manager. On request handle can listen for up to 15 events. There is no restriction on the number of request handles an application can activate.
- A list of events to be notified about, enclosed in parentheses.

The application uses the request handle in a subsequent EVENT WAIT statement to signal that it is listening. The event manager uses the handles to determine which applications should be woken up for notification.

❖ Notification will fail if the list of event names in parentheses does not match event names posted by triggers or stored procedures.

Registering interest in an event

To register interest in a single event, use the following EVENT INIT syntax:

```
EXEC SQL  
EVENT INIT request_name (event_name);
```

event_name can be up to 15 characters, and can be passed as a constant string in quotes, or as a host-language variable.

For example, the following embedded Firebird application code creates a request named RESPOND_NEW that registers interest in the "new_order" event:

```
EXEC SQL  
EVENT INIT RESPOND_NEW ('new_order');
```

The next example illustrates how RESPOND_NEW might be initialized using a host-language variable, myevent, to specify the name of an event:

```
EXEC SQL  
EVENT INIT RESPOND_NEW (:myevent);
```

After an application registers interest in an event, it is not notified about an event until it first pauses execution with EVENT WAIT.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Registering interest in multiple events

Although an application can only wait on a single request handle at a time, it might want to listen for several different events. The EVENT INIT statement enables an application to specify a list of event names in parentheses, using the following syntax:

```
EXEC SQL  
EVENT INIT request_name (event_name [event_name ...]);
```

For example, the following embedded application code creates a request named RESPOND_MANY that registers interest in three events, "new_order," "change_order," and "cancel_order":

```
EXEC SQL  
EVENT INIT RESPOND_MANY ('new_order', 'change_order', 'cancel_order');
```

An application can register interest in special groupings of multiple events by using separate EVENT INIT statements with unique request handles. Nevertheless, it still can wait on only one request handle at a time.

Asynchronous signaling

The event notification mechanism described here is synchronous—the application literally waits in suspension for an event and continues only when the event occurs and it has something to respond to.

Applications which communicate with Firebird server through the API have the alternative to implement asynchronous event notification. An API call can be used to register interest in an event and an asynchronous trap (AST) function identified to receive event notification and take care of ensuring that it is responded to by the application. An application can thus continue other processing instead of suspending its work to wait for an event to occur.

For example, a stock brokering application requires constant access to the Stocks database to provide brokers with real-time information as prices fluctuate; but it also needs to watch particular stocks continuously and trigger off the appropriate Buy or Sell procedure when certain events occur.

The `isc_que_events()` function and the AST function provide asynchronous event handling for a set of events that has been set up in an events parameter buffer (EPB). The EPB is populated by a call to the

**Symbols** [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

isc_event_block() function. Applications that need to respond to more than 15 events can make multiple calls to *isc_event_block()*, specifying different EPBs and *event_list* buffers for each call.

The AST function

The AST (asynchronous trap) function is a module which you write for the purpose of being called by Firebird when an event of interest is posted and handling the callback processing consequent to the event.

The AST, which should be written to take three arguments, has to notify the application that it has been called. The *isc_event_block()* function accepts into its *isc_callback* parameter a pointer to the AST function and, into its *event_function_arg* parameter, a pointer to the first argument of the AST. This argument generally accepts event counts as they change. The other arguments of the AST take the length of the *events_list* buffer and a pointer to that buffer. The AST also needs to provide for setting some form of global flag in the application to inform it that it has been called by Firebird.



For more details on handling events though the Firebird API, refer to the *API Guide* (APIGuide.pdf) of the InterBase® 6 documentation set, available from Borland.

Listening for events with EVENT WAIT

An application does not receive notification about an event until it signals to the event manager that it is listening for it. To indicate its readiness to the event manager and to suspend its own processing until notification occurs, it must use the EVENT WAIT statement:

```
EXEC SQL
EVENT WAIT request_name;
• request_name must be the name of a request handle previously declared in an EVENT INIT statement.
```

The following statements register interest in an event, and wait for event notification:

```
EXEC SQL
EVENT INIT RESPOND_NEW ('new_order');
EXEC SQL
EVENT WAIT RESPOND_NEW;
```



Symbols **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

Upon execution of EVENT WAIT, application processing stops until the event manager sends a notification message to the application.

- ★ An application can contain more than one EVENT WAIT statement, but all processing stops when the first statement is encountered. Each time processing restarts, it stops when it encounters the next EVENT WAIT statement.

If one event occurs while an application is processing another, the event manager sends notification the next time the application returns to a wait state.

Responding to events

When event notification occurs, a suspended application resumes normal processing at the next statement following EVENT WAIT.

If an application has registered interest in more than one event with a single EVENT INIT call, the application must read the event array, *isc_event[]*. to determine which event occurred.

- In an embedded application, the event array is automatically created for an application during preprocessing. Each element in the array corresponds to an event name passed as an argument to EVENT INIT. The value of each element is the number of times that event occurred since execution of the last EVENT WAIT statement with the same request handle.
- In an API application, the equivalent information is extracted by calling the *isc_event_counts()* function.

- For more details on handling events though the Firebird API, refer to the *API Guide* (APIGuide.pdf) of the InterBase® 6 documentation set, available from Borland.

In the following embedded application code, an application registers interest in three events, then suspends operation pending event notification:



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

```
EXEC SQL
EVENT INIT RESPOND_MANY ( 'new_order' , 'change_order' , 'cancel_order' );
EXEC SQL
EVENT WAIT RESPOND_MANY;
```

When any of the “new_order,” “change_order,” or “cancel_order” events are posted and their controlling transactions commit, the event manager notifies the application and processing resumes. The following code illustrates how an application might test which event occurred:

```
for ( i = 0; i < 3; i++ )
{
if (isc_$event[i] > 0)
{
/* this event occurred, so process it */
. . .
}
```

Using RDB\$DB_KEY

(with acknowledgement to Claudio Valderrama, who researched this topic)

Firebird inherits from its InterBase® antecedents a proprietary feature which can provide some additional performance power to queries in some conditions. This is RDB\$DB_KEY—usually referred to simply as *db_key*—an internal cardinality key maintained by the database engine for its control of record versions. To some extent, it represents a row’s position in the table, inside the transaction context in which it is captured.

Size of RDB\$DB_KEY

For tables, RDB\$DB_KEY uses 8 bytes. For a view, it uses as many multiples of 8 bytes as there are underlying tables. For example, if a view joins three tables, its RDB\$DB_KEY uses 24 bytes. This is important if you are working with stored procedures and want to store RDB\$DB_KEY in a variable. You must use a CHAR(n) data type of the correct length.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

By default, db_keys are returned as hex values—two hex digits represent each byte, causing 16 hex digits to be returned for 8 bytes:

```
SQL> SELECT RDB$DB_KEY FROM MYTABLE;
RDB$DB_KEY
=====
000000B600000002
000000B600000004
000000B600000006
000000B600000008
000000B60000000A
```

What is RDB\$DB_KEY

The first lesson to learn is that RDB\$DB_KEY is a raw position, related to the database itself and not to physical addresses on disk. The second is that, although the numbers progress, they do not progress in a predictable sequence. Don't consider performing calculations involving their relative positions! The third lesson is that they are volatile—they change after a backup and subsequent restore.

Usefulness of RDB\$DB_KEY

Because a RDB\$DB_KEY marks the raw position of a row, it is faster for a search than even a primary key. If for some special reason a table has no primary key, or the indexes are inactive (so permitting the possibility that exact duplicates may exist), RDB\$DB_KEY is the only way to identify each row unequivocally.

Several kinds of statements run faster when moved into a stored procedure using a RDB\$DB_KEY—typically, updates and deletions with complex conditions. However, if the database pages needed are in main memory, the requested set is reasonably small and all of the rows are in close proximity to one another, the difference in access speed is likely to be negligible. For inserts—even huge batches—the RDB\$DB_KEY is of no avail, since there is no way to ascertain what the values will be.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Optimization of queries

The Firebird optimizer still has trouble with some statements. Performance problems are likely if you try to run something like the following against a huge table—

```
update tableA A
set sumfield = (select sum(B.valfield)
from tableB B where B.FK = A.PK)
where <condition..>
```

If you run the same operation often, it would be worth the effort to write a stored procedure doing

```
for select B.FK, sum(B.valfield) from tableB B
group by B.FK
into :b_fk, :sum_vf do
update tableA A set sumfield = :sum_vf
where A.PK= :b_fk and condition
```

Although speedier, it still has the problem that records in A have to be located by primary key each time a new pass of the FOR...DO loop happens.

Some people claim better results with this alien syntax:

```
for select [B.FK,] sum(B.valfield), A.RDB$DB_KEY
from tableB B join tableA A on A.PK=B.FK
where <condition>
group by B.FK
into [:b_fk,] :sum_vf, :dbk do
update tableA A set sumfield = :sum_vf
where A.RDB$DB_KEY = :dbk
```

- ★ The two items shown with square brackets (not part of the syntax) are not required by Firebird. Older InterBase® versions might insist that the field used in the GROUP BY must be present in the SELECT clause. The second bracketed item depends on the first, because of the requirement for all output columns described in the SELECT to be received into variables, in order, after the INTO clause.
- ★ The variable *dbk*, holding the RDB\$DB_KEY value, must be defined as char(8) when you are dealing with a table.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Benefits

The benefits of this approach are:

- 1 filtering of the common records for A and B is efficient where optimizer can make a good filter from the explicit join.
- 2 the WHERE follows, to make additional filtering *before* the update checks its own condition.
- 3 rows from the dependent table (A) are located by raw db_key values, extracted at the time of the join, making the search faster than looking through the primary key.

Because, usually, an explicit query plan cannot be utilized in insertions, updates or deletes, procedures are really the only way to avoid slow performance.

Inserting

Since insertion does not involve locates, the simplest insert operations—for example, reading constant values from an import set in an external table—are not affected by the need to locate keys.

Not every insert statement's VALUES input set is obtained so simply, however. It can be a very complicated set of values derived from expressions, joins or aggregations. In addition, in a stored procedure, an INSERT operation may well be an operation branched into the ELSE sub-clause of an IF (EXISTS(...)) predicate. For example,

```
...
ELSE BEGIN
  insert into tableA
  select C.pkey, sum(B.a), avg(B.b), count(distinct C.c)
  from tableB B join tableC C on B.a = C.h and C.e = somevalue
  where C.pkey not in (select pkey from tableA)
  group by C.pkey;
END
...
```

You can compare the performance of the IN() and EXISTS() predicates by substituting

```
where C.pkey not in (select pkey from A)
```

with

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

where exists(select pkey from tableA A where A.pkey = C.pkey)

Implementing this in a stored procedure:

```
for select C.pkey, sum(B.a), avg(B.b), count(distinct C.c),
from tableB B join tableC C on B.a = C.h and C.e = somevalue
group by C.pkey,
into :c_key, :sum_a, :avg_b, :count_c do
begin
select A.RDB$DB_KEY from tableA A where A.pkey = :c_key
into :dbk;
/* if (sum_a is NULL) then sum_a = 0;
if (avg_b is NULL) then avg_b = 0; */
if (:dbk is NULL)
then insert into
tableA(pkey, summary, average, count_of_distinct)
values(:c_key, :sum_a, :avg_b, :count_c);
else update tableA A set
summary = summary + :sum_a, average = average + :avg_a,
count_of_distinct = count_of_distinct + :count_c
where A.RDB$DB_KEY = :dbk;
end
```

- ★ In the case that B.a and B.b were allowed to be NULL, a procedure such as this would be the only native way to handle that condition before the insert or update. The purpose of the two lines that are commented out here is to provide for this situation if it is allowed.

Duration of validity

By default, a db_key is valid only for the duration of the current transaction. A commit or rollback will cause the RDB\$DB_KEY values you had to become inaccurate. If you are using CommitRetaining, the transaction context is retained, blocking garbage collection. Hence, the RDB\$DB_KEY values remain valid until a "hard" commit occurs.

Once a "hard commit" of your transaction occurs, those RDB\$DB_KEY values become undependable because another transaction might have deleted a record that was isolated inside your transaction's context and was

**Symbols** [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

thus considered to exist. Any of your RDB\$DB_KEY values might now point to a non-existent row. If there is a long interval between the moment when your transaction began and when your work completes, you should check that the row has not been changed or locked by another transaction in the meantime.

The default duration of RDB\$DB_KEY values can be changed at connection time, by using the API or, in certain development environments, a connectivity class that surfaces this API item—for example, IB Objects, if you are developing in Borland Delphi or C++Builder. It is even possible to specify that the values be retained for the entire duration of the database session. However, this is not recommended in a highly interactive environment, since it will disable garbage collection, with the unwanted side-effect of causing your database file to grow at an alarming rate and its performance to slow down, or stop altogether.

Qualifying RDB\$DB_KEY in the output set

```
select RDB$DB_KEY from table
```

will work without problems, but

```
select RDB$DB_KEY, column_name from table
```

does not work because it need explicits table qualification:

```
select table.RDB$DB_KEY, field from table
```

❖ When the table name is long or convoluted, or there are multiple tables, use table aliases in the interests of clear code—see [Table aliases](#) on p. 132 of chapter 9.

❖ Recall, too, that all tables maintain their own distinct RDB\$DB_KEY columns and use 8 bytes (more in views). If you use RDB\$DB_KEY on multiple columns, be very careful to qualify each one accurately.

RDB\$DB_KEY as a JOIN criterion

RDB\$DB_KEY can **not** be used across tables. The RDB\$DB_KEY of one table has no possible relationship with that of another, except in self-referencing joins. RDB\$DB_KEY records a raw record position and no two tables use the same physical space in the database.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

CHAPTER 26

Working with UDFs and Blob Filters

Firebird “travels light” with respect to built-in functions. Instead of a vast library of esoteric functions to weigh down the server, Firebird provides for developers to select—and, if necessary to define—their own libraries of external functions to suit the calculation and expression requirements in their databases.

User-defined functions—known to all as UDFs—add great flexibility to your database environment. This chapter introduces the topic and many of the issues concerned with writing, compiling and using them. It also introduces the technique of writing and using BLOB filters—special UDFs that can be used directly in SQL to store and convert undefined types of binary objects to their native or alternative formats.

About UDFs

UDFs—*user-defined functions*—are host-language programs for performing custom data transformations in Firebird SQL statements and code modules. Like the standard, built-in SQL functions, UDFs can be designed to do conversions or calculations that are either complex or impossible to do with the SQL language. Possibilities include statistical, string, date and mathematical functions. UDFs are server-side extensions to the Firebird server that are declared to databases and executed in the context of the server process.

Existing libraries

Firebird ships with either one or two pre-built libraries of UDFs, described in the section *User-defined Functions* in the *Firebird Reference Guide*. The default installations place these shared object libraries in the *./udf* directory beneath the Firebird root directory. On Windows, the libraries are dynamic link library (.dll) files. On Linux/UNIX they are compiled dynamic library objects named with the extension ‘.so’ or with no extension at all in some cases.

The libraries are:

**Symbols** [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

- **ib_udf**—a library of useful, basic functions which Firebird inherited with its InterBase® ancestry. This library needs to call some memory utilities which are located in a companion shared object name **ib_util**, located in the **./bin** directory. This library passes parameters either *by value* or *by reference*, in the conventional InterBase® style.
- **FBUDF.dll**—by Claudio Valderrama, initially available only for Windows servers, passes parameters *by Firebird descriptor*, considered a more robust way to ensure that internal errors do not occur as a result of memory allocation and type conversion errors.

Also freely available are several public domain UDF libraries, including **FreeUDFLib**, which was originally written in Delphi™ by Gregory Deatz. FreeUDFLib contains a vast number of string, math, BLOB and date functions. This library has been maintained, corrected and augmented by several people over the years. In 2000, Robert Schieck issued a Linux version in which some functions work satisfactorily. Others have distributed, or continue to work on Kylix™ conversions of the original Pascal for more robust and comprehensive use on UNIX-compatible platforms.

❖ Care should be taken to obtain a version of this library that works properly with Dialect 3 date and time types. Such a version, for Windows, is available from <http://www.cvalde.com>.

❖ For a list of existing UDF libraries, see <http://www.ibphoenix.com>.

You can access UDFs and BLOB filters by using them anywhere that a built-in SQL function can be used—as SQL expressions in statements through **isql** or a query in an application or host-language program. UDFs can also return values to variables or SQL expressions in stored procedure and trigger bodies.

Developing a UDF

Overview

UDF libraries are standard shared libraries that are dynamically loaded by the database at runtime. You can create UDF libraries on any platform that is supported by Firebird. To use the same set of UDFs with databases running on different platforms, create separate libraries on each platform where the databases reside. UDFs run on the server where the database resides.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

★ A *library*, in this context, is a shared object that typically has a .dll extension on Windows platforms, a .so extension on UNIX and Solaris, and a .sl extension on HP-UX.

Book The Firebird ./examples directory contains sample makefiles (makefile.bc and makefile.msc on Windows systems, makefile on UNIX) that build the ib_udf function library from ib_udf.c.

Creating and implementing a UDF is a four-step process:

- 1 Write the function in any programming language that can create a shared library. A function can take a limited number of entry parameters but it must return one and only one result. *Functions written in Java are not supported.*
- 2 Compile the function and link it to a dynamically linked library (DLL).
- 3 Place the library and any symbolic links required into the appropriate disk locations on the server machine.
- 4 Use DECLARE EXTERNAL FUNCTION to declare each individual UDF to each database in which you need to use it.

Writing a function module

To create a user-defined function (UDF), code the UDF in a host language, then build a shared function library that contains the UDF. In the C language, a UDF is written like any standard function. The UDF can provide for up to 10 input parameters, and must return one and only one C data value as its result.

A source code module can define one or more functions and can use typedefs defined in the Firebird **ibase.h** header file provided **ibase.h** is included in the compilation.

Specifying parameters

A UDF can accept up to 10 parameters corresponding to any Firebird datatype. Array elements cannot be passed as parameters. If a UDF returns a blob, the number of input parameters is restricted to nine. All parameters are passed to the UDF by reference.

Programming language datatypes specified as parameters must be capable of handling corresponding Firebird datatypes. For example, the C function declaration for FN_ABS() accepts one parameter of type double. The expectation is that when FN_ABS() is called, it will be passed a datatype of DOUBLE PRECISION by Firebird.



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

BLOB parameters

UDFs that accept BLOB parameters require special data structure for processing. A BLOB is passed by reference to a BLOB UDF structure. For more information about the BLOB UDF structure, see [Writing a blob UDF](#) on page 586.

Specifying a return value

A UDF can return any value that can be translated into a Firebird data type, including BLOB, but it cannot return arrays of any data type. For example, the C function declaration for FN_ABS() returns a value of type double, which corresponds to the Firebird DOUBLE PRECISION datatype.

By default, return values are passed by reference. Numeric values can be returned by reference or by value. To return a numeric parameter by value, include the optional BY VALUE reserved word after the return value when declaring a UDF to a database.

A UDF that returns a BLOB does not actually define a return value. Instead, a pointer to a structure describing the blob to return must be passed as the last input parameter to the UDF.

Character datatypes

UDFs are written in a host language and therefore take host-language datatypes for both their parameters and their return values. However, when a UDF is declared, Firebird must translate it to a SQL datatype or to a CSTRING type of a specified maximum byte length. CSTRING is used to translate parameters of CHAR and VARCHAR datatypes into a null-terminated C string for processing, and to return a variable-length, null-terminated C string to Firebird for automatic conversion to CHAR or VARCHAR.

- ❖ When declaring to a database a UDF that returns a C string, CHAR or VARCHAR, the keyword FREE_IT must be included in the declaration in order to free the memory used by the return value.

Calling conventions

The calling convention determines how a function is called and how the parameters are passed. The '*callee function*' (the function receiving the function call) must match the caller function's calling convention. Because Firebird uses the *STDCALL* calling convention, all UDFs must use the same calling convention. To make a function use the *STDCALL* calling convention the *_stdcall* reserved words must be added to the function declaration.

Here is an example function that specifies the *STDCALL* calling convention:



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

```
ISC_TIMESTAMP* __stdcall addmonth(ISC_TIMESTAMP *preTime)
{
    // body of function here
}
```

Thread-safe UDFs

In SuperServer implementations of Firebird, the server runs as a single multi-threaded process. This means that you must take some care in the way you allocate and release memory when coding UDFs and also in the way you declare UDFs.

Several issues need to be considered when handling memory in the single-process, multi-thread architecture:

- UDFs must avoid static variables in order to be thread safe. You can use static variables only if you can guarantee that only one user at a time will be accessing UDFs, since users running UDFs concurrently will conflict when they step on the same static memory space.
 - ❖ If you cannot avoid returning a pointer to static data, you must *not* use FREE_IT.
- UDFs must allocate memory using *ib_util_malloc* rather than static arrays in order to be thread-safe.
- Memory allocated dynamically is not automatically released, since the process does not end. You must use the FREE_IT reserved word when you declare the UDF to the database (DECLARE EXTERNAL FUNCTION).

Multi-process version (not thread-safe: do not use for SuperServer)

In the following example for user-defined function FN_LOWER(), the array must be global to avoid going out of context:

```
char buffer[256];
char *fn_lower(char *ups)
{
    . . .
    return (buffer);
}
```



Symbols **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

Thread-safe version

In the following version, the Firebird engine will free the buffer if the UDF is declared using the FREE_IT reserved word. Notice that this example uses Firebird's *ib_util_malloc()* function to allocate memory:

```
char *fn_lower(char *ups)
{
    char *buffer = (char *) ib_util_malloc(256);
    . . .
    return (buffer);
}
```

Making UDFs thread-safe

The procedure for allocating and freeing memory for return values in a fashion that is both thread safe and compiler independent is as follows:

- 1 In the UDF code, use Firebird's *ib_util_malloc()* function to allocate memory for return values.
 - This function is in **./lib/ib_util.dll** on Windows, **./lib/ib_util.so** on Solaris, and **./lib/ib_util.s1** on HP-UX.
- 2 Use the FREE_IT keyword in the RETURNS clause when declaring a function that returns dynamically allocated objects. For example:

```
DECLARE EXTERNAL FUNCTION lowers VARCHAR(256)
RETURNS CSTRING(256) FREE_IT
ENTRY POINT 'fn_lower' MODULE_NAME 'ib_udf'
```

★ Firebird's FREE_IT keyword allows Firebird users to write thread-safe UDF functions without memory leaks.

- 3 Memory must be released by the same runtime library that allocated it.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Compiling and linking a function module

When a UDF module is ready, compile it in a normal fashion into object or library format.

- Include `ibase.h` in the source code if you use typedefs defined in the Firebird `ibase.h` header file. All “include” (*.h) libraries are in the `./SDK/include` directory.
- Link to `gds32.dll` if you use calls to Firebird library functions.

Microsoft Visual C/C++

Link with `./SDK/lib/ib_util_ms.lib` and include `./SDK/include/ib_util.h`

Use the following options when compiling applications with Microsoft C++:

TABLE 26-1 Microsoft C compiler options

Option	Action
<code>c</code>	Compile without linking (DLLs only)
<code>Zi</code>	Generate complete debugging information
<code>DWIN32</code>	Defines “WIN32”
<code>D_MT</code>	Use a multi-thread, statically-linked library

Borland C++

Link with `./SDK/lib/ib_util.lib` and include `./SDK/include/ib_util.h`

Delphi

Use `./SDK/include/ib_util.pas`.

Examples

The following commands use the Microsoft compiler to build a DLL that uses Firebird:



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

```
cl -c -Zi -DWIN32 -D_MT -LD udf.c  
lib -out:udf.dll -def:funclib.def -machine:i586 -subsystem:console  
link -DLL -out:funclib.dll -DEBUG:full,mapped -DEBUGTYPE:CV  
-machine:i586 -entry:_DllMainCRTStartup@12 -subsystem:console  
-verbose udf.obj udf.exp gds32_ms.lib ib_util_ms.lib crtdll.lib
```

This command builds a Firebird executable using the Microsoft compiler:

```
cl -Zi -DWIN32 -D_MT -MD udftest.c udf.lib gds32_ms.lib  
ib_util_ms.lib crtdll.lib
```

See the makefiles (**makefile.bc** and **makefile.msc** on Windows systems, **makefile** on UNIX) in the Firebird **./examples** subdirectory for details on how to compile a UDF library.

For examples of writing thread-safe UDFs, see **./examples/UDFib_udf.c**. This file contains the source code for the **ib_udf** library.

Modifying a UDF library

To add a UDF to an existing UDF library on a platform:

- Compile the UDF according to the instructions for the platform.
- Include all object files previously included in the library and the newly-created object file in the command line when creating the function library.
 On some platforms, object files can be added directly to existing libraries. For more information, consult the platform-specific compiler and linker documentation.

To delete a UDF from a library, follow the linker's instructions for removing an object from a library.

Deleting a UDF from a library does not remove its declaration in the database.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Declaring a UDF to a database

Once a UDF has been written and compiled into a library and installed into the external_function_directory, it must be declared to the database using the DECLARE EXTERNAL FUNCTION statement in order to be used as an SQL function.

As soon as function is declared, the library in which it is contained will be loaded automatically at run time the first time an application calls any function it contains, in any database which uses it.

A UDF library is available for use by any database running on the server which which is associated. However, it is necessary to declare each function to each database in which it will be used. Each function in a library must be declared separately, but needs to be declared only once to each database.

Declaring a UDF to a database informs the database about its location and properties:

- The UDF name as it will be used in embedded SQL statements
- The number and datatypes of its arguments
- The return datatype
- The name of the function as it exists in the UDF module or library
- The name of the library that contains the UDF

You can use **isql**, another interactive SQL tool, or a script to declare UDFs.

Declaration syntax

```
DECLARE EXTERNAL FUNCTION name [datatype | CSTRING (int)  
[, datatype | CSTRING (int) ...]]  
RETURNS {datatype [BY VALUE] | CSTRING (int)} [FREE_IT]  
[RETURNS PARAMETER n]  
ENTRY_POINT 'entryname'  
MODULE_NAME 'modulename';
```



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

TABLE 26-2 DECLARE EXTERNAL FUNCTION arguments

Argument	Description
<i>name</i>	Name of the UDF to use in SQL statements; can be different from the name of the function specified after the ENTRY_POINT reserved word
<i>datatype</i>	Datatype of an input or return parameter <ul style="list-style-type: none">• All input parameters are passed to a UDF by reference• Return parameters can be passed by value• Cannot be an array element
RETURNS	Specifies the return value of a function
BY VALUE	Specifies that a return value should be passed by value rather than by reference
CSTRING (<i>int</i>)	Specifies a UDF that returns a null-terminated string <i>int</i> bytes in length
FREE_IT	Frees memory of the return value after the UDF finishes running <ul style="list-style-type: none">• Use only if the memory is allocated dynamically in the UDF• See also the UDF chapter in the <i>Developer's Guide</i>
RETURNS PARAMETER <i>n</i>	Specifies that the function returns the <i>n</i> th input parameter; required for returning Blobs
' <i>entryname</i> '	Quoted string specifying the name of the UDF in the source code and as stored in the UDF library



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

TABLE 26–2 DECLARE EXTERNAL FUNCTION arguments (*continued*)

Argument	Description
<i>modulename</i>	Quoted file specification identifying the library that contains the UDF <ul style="list-style-type: none">The library must reside on the serverOn any platform, the module can be referenced with no path name if it is in <code>./UDF</code> or <code>./int1</code> or in the external_function_directory path listIf you do not place the library in <code>./UDF</code> or <code>./int1</code> you <i>must</i> include its location in Firebird's configuration file using the <code>EXTERNAL_FUNCTION_DIRECTORY</code> parameterIt is not desirable to supply the file extension to the module name, since it will cause your database to be platform-dependent.See UDF library placement for more about how the operating system finds the UDF library

Declaring UDFs with FREE_IT

Firebird's `FREE_IT` reserved word allows Firebird users to write thread-safe UDF functions without memory leaks.

Whenever a UDF returns a value by reference to dynamically allocated memory, you must declare it using the `FREE_IT` reserved word in order to free the allocated memory.

Note You *must not* use `FREE_IT` with UDFs that return a pointer to static data, as in the "multi-process version" example on page 576.

The following code shows how to use this reserved word:

```
DECLARE EXTERNAL FUNCTION lowers VARCHAR(256)
RETURNS CSTRING(256) FREE_IT
ENTRY POINT 'fn_lower' MODULE_NAME 'ib_udf'
```



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

UDF library placement

Place UDF libraries in the UDF directory. Or, specify additional UDF directories by adding entries to Firebird configuration file `ib_config` (on Windows) or `isc_config` (on UNIX).

To specify a location in a directory on your server for UDF libraries in a configuration file:

On Windows:

```
EXTERNAL_FUNCTION_DIRECTORY "D:\Mylibraries\Firebird"
```

On UNIX:

```
EXTERNAL_FUNCTION_DIRECTORY "/usr/local/lib/Mylibraries/Firebird"
```

Firebird finds the functions once you have declared them with `DECLARE EXTERNAL FUNCTION`. You should not specify a path in the declaration.

- ❖ It is not sufficient to include a complete path name for the module in the `DECLARE EXTERNAL FUNCTION` statement. You *must* list the path in the Firebird configuration file if it is other than `./UDF` or `./intl`.

Security of UDF libraries

Because UDF libraries are external files under operating system filesystem control, they will be vulnerable to accidental or malicious interference if not properly secured. The Firebird support network strongly recommends that you place your compiled libraries in directories that are dedicated to Firebird libraries and protected from unauthorized file-level access.

- ❖ For server node security reasons, it is also recommended that you do not locate Firebird libraries in system directories such as `C:\winnt40\system32` or `/usr/lib`.

Examples—UDF declaration

The following statement declares the `TOPS()` UDF to a database:

```
DECLARE EXTERNAL FUNCTION TOPS  
CHAR(256), INTEGER, BLOB  
RETURNS INTEGER BY VALUE  
ENTRY_POINT 'TE1' MODULE_NAME 'TM1.DLL';
```

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

This example does not need the FREE_IT reserved word because only the cstring, CHAR, and VARCHAR return types require memory allocation. The module must be in Firebird's UDF directory or in an EXTERNAL_FUNCTION_DIRECTORY that is named in the configuration file.

The following **isql** script declares three UDFs, ABS(), DATEDIFF(), and TRIM(), to the **employee.gdb** database:

```
CONNECT 'employee.gdb';
DECLARE EXTERNAL FUNCTION ABS
DOUBLE PRECISION
RETURNS DOUBLE BY VALUE
ENTRY_POINT 'fn_abs' MODULE_NAME 'ib_udf';
DECLARE EXTERNAL FUNCTION DATEDIFF
DATE, DATE
RETURNS INTEGER
ENTRY_POINT 'fn_datediff' MODULE_NAME 'ib_udf';
DECLARE EXTERNAL FUNCTION TRIM
SMALLINT, CSTRING(256), SMALLINT
RETURNS CSTRING(256) FREE_IT
ENTRY_POINT 'fn_trim' MODULE_NAME 'ib_udf';
COMMIT;
```

★ Note that no extension is supplied for the module name. This creates a portable module. Windows machines add a **.dll** extension automatically.

Finding UDF declaration details

By tradition, when developers create UDF libraries, they distribute a script containing all of the declarations for the functions in the library. Usually, the file is named *udf_lib_name.sql*, where *udf_lib_name* is the file-name stem used for the library file itself. For example, in the Firebird ./examples directory, you will find scripts named *ib_udf.sql* and *FBUDF.sql*.

★ Unfortunately, FreeUDFLib does not follow this convention. Its declaration script is named *ext_funcs.sql*.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Calling a UDF

After a UDF is created and declared to a database, it can be used in SQL statements wherever a built-in function is permitted. To use a UDF, insert its name in a SQL statement at an appropriate location, and enclose its input arguments in parentheses.

For example, the following DELETE statement calls the ABS() UDF as part of a search condition that restricts which rows are deleted:

```
DELETE FROM CITIES  
WHERE ABS (POPULATION - 100000) > 50000;
```

UDFs can also be called in stored procedures and triggers, as with built-in functions, to transform a value into a variable. For example,

```
...  
DECLARE VARIABLE absolute_val integer;  
...  
absolute_val = ABS (POPULATION - 100000);  
...
```

Calling a UDF with SELECT

In SELECT statements, a UDF can be used in a select list to specify data retrieval, or in the WHERE clause search condition.

The following statement uses ABS() to guarantee that a returned column value is positive:

```
SELECT ABS (JOB_GRADE) FROM PROJECTS;
```

The next statement uses DATEDIFF() in a search condition to restrict rows retrieved:

```
SELECT START_DATE FROM PROJECTS  
WHERE DATEDIFF (:today, START_DATE) > 10;
```

Calling a UDF with INSERT

In INSERT statements, a UDF can be used in the comma-delimited list in the VALUES clause.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

The following statement uses TRIM() to remove leading blanks from *firstname* and trailing blanks from *lastname* before inserting the values of these host variables into the EMPLOYEE table:

```
INSERT INTO EMPLOYEE(FIRST_NAME, LAST_NAME, EMP_NO, DEPT_NO, SALARY)
VALUES (TRIM (0, ' ', :firstname), TRIM (1, ' ', :lastname),
:empno, :deptno, greater(30000, :est_salary));
```

Calling a UDF with UPDATE

In UPDATE statements, a UDF can be used in the SET clause as part of the expression assigning column values. For example, the following statement uses TRIM() to ensure that update values do not contain leading or trailing blanks:

```
UPDATE COUNTRIES
SET COUNTRY = TRIM (2, ' ', COUNTRY);
```

Calling a UDF with DELETE

In DELETE statements, a UDF can be used in a WHERE clause search condition:

```
DELETE FROM COUNTRIES
WHERE ABS (POPULATION - 100000) < 50000;
```

Writing a blob UDF

A blob UDF differs from other UDFs, because pointers to blob control structures are passed to the UDF instead of references to actual data. A blob UDF cannot open or close a blob, but instead invokes functions to perform blob access.

Creating a blob control structure

A blob control structure is a C struct, declared within a UDF module as a `typedef`. Programmers must provide the control structure definition, which should be defined as follows:



Symbols **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

```
typedef struct blob {  
    void (*blob_get_segment) ();  
    isc_blob_handle blob_handle;  
    long number_segments;  
    long max_seglen;  
    long total_size;  
    void (*blob_put_segment) ();  
} *Blob;
```

TABLE 26-3 Fields in the blob control structure

Field	Description
<i>blob_get_segment</i>	The first field in the blob struct, <i>blob_get_segment</i> , is a pointer to a function that is called to read a segment from a blob, if a blob is passed to the UDF. The function takes four arguments: <ul style="list-style-type: none">• a blob handle• the address of a buffer for the blob segment• the size of the buffer• the address of a variable to hold the size of the blob segment. If blob data is not read by the UDF, set <i>blob_get_segment</i> to NULL.
<i>blob_handle</i>	The second field in the blob struct is required. It is a blob handle that uniquely identifies a blob passed to a UDF or returned by it.
<i>number_segments</i>	For blob data passed to a UDF, <i>number_segments</i> specifies the total number of segments in the blob. Set this value to NULL if blob data is not passed to a UDF.

**Symbols** [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)TABLE 26-3 Fields in the blob control structure (*continued*)

Field	Description
<i>max_seglens</i>	For blob data passed to a UDF, <i>max_seglens</i> specifies the size, in bytes, of the largest single segment passed. Set this value to NULL if blob data is not passed to a UDF.
<i>total_size</i>	For blob data passed to a UDF, <i>total_size</i> specifies the actual size, in bytes, of the blob as a single unit. Set this value to NULL if blob data is not passed to a UDF.
<i>blob_put_segment</i>	The last field in the blob struct, <i>blob_put_segment</i> , is a pointer to a function that is called to write a segment to a blob if one is being returned by the UDF. The function takes three arguments: a blob handle, the address of a buffer containing the data to write into the blob, and the size, in bytes, of the data to write. If blob data is not read by the UDF, set <i>blob_put_segment</i> to NULL.

Declaring a Blob UDF

A Blob UDF is declared to the database using DECLARE EXTERNAL FUNCTION like any non-Blob UDF. Use the RETURNS PARAMETER *n* statements to specify which input blob is to be returned. For example, the return the third input blob, specify RETURNS PARAMETER 3. The following statement declares the Blob UDF, Blob_PLUS_Blob, to a database:

```
DECLARE EXTERNAL FUNCTION Blob_PLUS_Blob
    Blob,
    Blob,
    Blob
RETURNS PARAMETER 3
ENTRY_POINT 'blob_concatenate' MODULE_NAME 'ib_udf';
COMMIT;
```



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

A blob UDF example

The following code creates a UDF, *blob_concatenate()*, that appends data from one blob to the end of another blob to return a third blob consisting of concatenated blob data. You can use *malloc()* rather than *ib_util_malloc()* when you free the memory in the same function where you allocate it.

```
/* Blob control structure */
typedef struct blob {
void (*blob_get_segment) ();
int *blob_handle;
long number_segments;
long max_seglen;
long total_size;
void (*blob_put_segment) ();
} *Blob;

extern char *isc_$alloc();
#define MAX(a, b) (a > b) ? a : b
#define DELIMITER "-----"

void blob_concatenate(Blob from1, Blob from2, Blob to)
/* Note Blob to, as final input parameter, is actually for output! */
{
char *buffer;
long length, b_length;

b_length = MAX(from1->max_seglen, from2->max_seglen);
buffer = malloc(b_length);
```

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

```
/* write the from1 Blob into the return Blob, to */
while ((*from1->blob_get_segment) (from1->blob_handle, buffer,
b_length, &length))
(*to->blob_put_segment) (to->blob_handle, buffer, length);

/* now write a delimiter as a dividing line in the blob */
(*to->blob_put_segment) (to->blob_handle, DELIMITER,
sizeof(DELIMITER) - 1);

/* finally write the from2 Blob into the return Blob, to */
while ((*from2->blob_get_segment) (from2->blob_handle, buffer,
b_length, &length))
(*to->blob_put_segment) (to->blob_handle, buffer, length);

/* free the memory allocated to the buffer */
free(buffer);
}
```

The ib_udf libraries

ib_udf

Firebird provides a number of frequently needed functions in the **ib_udf.dll** on Windows platforms and **ib_udf** on UNIX platforms. These UDFs are located in **./udf** and are all implemented using the standard C library. This section describes each UDF and provides its declaration.

The script **ib_udf.sql**, in the **./examples** directory that declares all of the functions listed below. If you want to declare only a subset of these, copy and edit the script file.

- ❖ Several of these UDFs must be called using the FREE_IT keyword if—and only if—they are written in thread-safe form, using *malloc* to allocate dynamic memory.



Symbols **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

★ When trigonometric functions are passed inputs that are out of bounds, they return zero rather than NaN.

A list of the functions supplied in the ib_udf library follows. See the UDF chapter of the *Firebird Reference Guide* for more details about these functions.

Function	Description	Inputs	Outputs
ABS()	Absolute value	Double precision	Double precision
ACOS()	Arc cosine	Double precision	Double precision
ASCII_CHAR()	Return character based on ASCII code	Integer	Char(1)
ASCII_VAL()	Return ASCII code for given character	Char(1)	Integer
ASIN()	Arc sine	Double precision	Double precision
ATAN()	Arc tangent	Double precision	Double precision
ATAN2()	Arc tangent divided by second argument	Double precision, Double precision	Double precision
BIN_AND()	Bitwise AND operation	Integer	Integer
BIN_OR()	Bitwise OR operation	Integer	Integer
BIN_XOR()	Bitwise XOR operation	Integer	Integer
CEILING()	Round up to nearest whole value	Double precision	Double precision
COS()	Cosine	Double precision	Double precision
COSH()	Hyperbolic cosine	Double precision	Double precision
COT()	Cotangent	Double precision	Double precision
DIV()	Integer division	Integer	Integer

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

Function	Description	Inputs	Outputs
FLOOR()	Round down to nearest whole value	Double precision	Double precision
LN()	Natural logarithm	Double precision	Double precision
LOG()	Logarithm of the first argument, by the base of the second argument	Double precision, Double precision	Double precision
LOG10()	Logarithm base 10	Double precision	Double precision
LOWER()	Reduce all upper-case characters to lower-case	Cstring(80)	Cstring(80)
LTRIM()	Strip preceding blanks	Cstring(80)	Cstring(80)
MOD()	Modulus operation between the two arguments	Integer, Integer	Integer
PI()	Return the value of π	—	Double precision
RAND()	Return a random value	—	Double precision
RTRIM()	Strip trailing blanks	Cstring(80)	Cstring(80)
SIGN()	Return -1, 0, or 1	Double precision	Integer
SIN()	Sine	Double precision	Double precision
SINH()	Hyperbolic sine	Double precision	Double precision
SQRT()	Square root	Double precision	Double precision
STRLEN()	Length of string	Cstring(32767)	Integer
SUBSTR()	Substring, starting at position equal to second argument, with length equal to third argument	Cstring(80), Smallint, Smallint	Cstring(80)



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Function	Description	Inputs	Outputs
SUBSTRLEN()	Returns a string of size <length> starting at <pos>. The length of the string will be the lesser of <length> or the number of characters from <pos> to the end of the input string.	Cstring(80), Smallint, Smallint	Cstring(80)
TAN()	Tangent	Double precision	Double precision
TANH()	Hyperbolic tangent	Double precision	Double precision

FBUDF.dll

The FBUDF library—initially available only for Windows servers—is the first UDF library to pass arguments to and results from UDFs by Firebird descriptor, rather than by value or by reference. This approach allows NULL to be passed and also dispenses with the need to explicitly allocate and free memory space that is external to the Firebird engine.

A list of the functions supplied in the FBUDF.dll library follows. See the UDF chapter of the *Firebird Reference Guide* for more details about these functions.

Function	Description	Inputs	Outputs
iNvl()	'Numerical NVL'—Similar to NVL() function in Oracle: the first input is a constant, the second is the value to be returned if the first input is NULL	Integer Integer	Integer
sNvl()	'String NVL'—Similar to NVL() function in Oracle: the first input is a constant, the second is the value to be returned if the first input is NULL	Varchar(100) Varchar(100)	Varchar(100)

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

Function	Description	Inputs	Outputs
iNullIf()	Returns NULL if both inputs are equivalent or the result of the first expression if they are not.	Integer, Integer or Numeric(18,4), Numeric(18,4)	Integer Numeric(18,4)
sNullIf()	Returns NULL if both inputs are equivalent or the result of the first expression if they are not.	Varchar(100) Varchar(100)	Varchar(100) Varchar(100)
DOW()	Returns the day of the week on which the submitted timestamp falls	Timestamp	Varchar(15)
SDOW()	Returns the day of the week, abbreviated, on which the submitted timestamp falls	Timestamp	Varchar(5)
right()	Like rstring in Basic—returns the rightmost n characters of a string	Varchar(100) integer	Varchar(100)
addDay()	Adds n days to the submitted timestamp	Timestamp Integer	Timestamp
addWeek()	Adds n weeks to the submitted timestamp	Timestamp Integer	Timestamp
addMonth()	Adds n months to the submitted timestamp	Timestamp Integer	Timestamp
addYear()	Adds n years to the submitted timestamp	Timestamp Integer	Timestamp
addMillisecond()	Adds n milliseconds to the submitted timestamp	Timestamp Integer	Timestamp



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Function	Description	Inputs	Outputs
addSecond()	Adds n seconds to the submitted timestamp	Timestamp Integer	Timestamp
addMinute()	Adds n minutess to the submitted timestamp	Timestamp Integer	Timestamp
addHour()	Adds n hourss to the submitted timestamp	Timestamp Integer	Timestamp
getExactTimestamp()	Returns the current exact server time, to ten-thousandths of a second	<none>	Timestamp
truncate()	Takes scaled (exact-precision) numerics of any range (up to 9 in Dialect 1 or up to 19 in Dialect 3) and returns the whole-number portion. Does not work with float or double types.	Integer or Numeric(18)	Numeric(18)
round()	Takes 32-bit and 64-bit integers respectively, accepting scaled (exact-precision) numerics of any range (up to 9 in Dialect 1 or up to 19 in Dialect 3) and returning the nearest whole number. You cannot specify the number of decimal places.	Integer or Numeric(18,4)	Numeric(18,4)
string2blob()	Converts a Char or a Varchar type to a BLOB	Varchar(300)	BLOB



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

BLOB filters

BLOB filters are special utility programs written to convert data from one BLOB subtype to another. BLOB filters can be accessed from any client via SQL statements.

To implement BLOB filters, follow these steps:

- 1 Write the filters and compile them into object code.
- 2 Create a shared filter library (a dll object on Windows, a shared object on UNIX-compatible platforms).
- 3 Make the filter library available to Firebird at run time.
- 4 Declare the filters to the database using `DECLARE FILTER`.
- 5 Define columns in tables to store the typed BLOBS.
- 6 Write an application that requests filtering.

BLOB subtypes and filters can be used to encapsulate a useful range of processing tasks. For example, you can define pairs of BLOB subtypes to manage regular conversions required by your applications. The following scenarios are typical:

- One SUB_TYPE to hold compressed data and another to hold decompressed data. Write BLOB filters for expanding and compressing BLOB data.
- One SUB_TYPE to hold generic application code and others to hold system-specific code. Write BLOB filters that add the necessary system-specific variations to the generic code.
- One SUB_TYPE to hold XML-formatted text and others to hold output-specific code—HTML, rich text, portable document format (PDF), UNIX ‘man’ files, word-processor formats. Write BLOB filters that perform the transformations for the output.

The transformation code in your filter routine can be as simple or complex as you need it to be. It can call other modules, if necessary, thus putting at your disposal the ability to capitalize on existing code.

 BLOB filter capability obviates the need to implement transformation features in the database engine and keep in step with the technology. For instance, implementing XML capability in the Firebird database engine would be a “fudge” compared to the flexibility of a well-designed BLOB filter scenario.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Invoking BLOB filters

Firebird can invoke BLOB filters in any of the following ways:

- through SQL function calls in statements in an application
- through a stored procedure or trigger
- interactively through isql.

Pre-defined BLOB filters

Firebird already uses several pre-defined blob filters, one of which kicks in automatically when a statement requests an untyped blob (SUB_TYPE 0) which contains text. The pre-defined BLOB SUB_TYPES have non-negative SUB_TYPE *identifiers*.

Using the standard Firebird text filters

Firebird includes a set of special internal Blob filters that convert from SUB_TYPE 0 to SUB_TYPE 1 (TEXT), and from SUB_TYPE 1 (TEXT) to SUB_TYPE 0. These filters can also convert BLOB data of any Firebird system SUB_TYPE, to SUB_TYPE 1 (TEXT) which may be useful for returning the contents of a column in a system table to a database management application.

When a text filter is being used to read data from a BLOB column, it modifies the standard Firebird behavior for supplying segments. Regardless of the actual nature of the segments in the BLOB column, the text filter enforces the rule that segments must end with a newline character (ASCII 10). The text filter returns all the characters up to and including the first newline as the first segment, the next characters up to and including the second newline as the second segment, and so on.

- ★ To convert any non-text SUB_TYPE to TEXT, declare its FROM SUB_TYPE as SUB_TYPE 0 and its TO SUB_TYPE as SUB_TYPE 1.

SUB_TYPE identifiers

The system table RDB\$FILTERS tracks information about BLOB filters. A system flag with a negative number indicates a *user-defined filter*. Two SUB_TYPE identifiers are stored: one to indicate which SUB_TYPE is input to the filter, the other to indicate which SUB_TYPE to output. It is this second value that Firebird's internal conversion mechanism uses to find the external code module (declaration parameter MODULE_NAME) where

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

you placed your conversion routine and the entry point (declaration parameter ENTRY_POINT) within that module where its code begins.

- ★ Positive SUB_TYPE identifiers are reserved for internal use.

Custom BLOB filters

You can define your own SUB-TYPEs and provide host language code for converting them.

A user-defined SUB-TYPE is designated by a negative integer value. The actual integer value that you use is arbitrary, as long as it is unique in the database and within the range -1 to -32,678. Firebird itself has no mechanism for validating data which is input to a particular sub_type—when asked to return a BLOB for which no SUB_TYPE has been defined, or one which has not been declared to the database, Firebird automatically uses a built-in ASCII BLOB filter. It is the responsibility of the client application to handle any validation problems before new BLOB data are stored.

The only issue, then, when using user-defined SUB-TYPEs, is to ensure that the same type of binary data is stored for every row in the column specified to be that user-defined SUB_TYPE.

Declaring a custom BLOB filter

Declare the filter to the database with the DECLARE FILTER statement. For example:

```
DECLARE FILTER BLOB_FORMAT  
INPUT_TYPE 1 OUTPUT_TYPE -99  
ENTRY_POINT 'Text_filter' MODULE_NAME 'Filter_99.dll';
```

- iben For more information about DECLARE FILTER syntax, refer to the Statement and Function Reference section of the *Firebird Reference Guide*.

Defining a column for a custom BLOB filter

To define a column to take binary data of the type recognized by your custom BLOB filter, use the following syntax:

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

```
CREATE TABLE table_name
(
  ...,
  column_name BLOB SUB_TYPE n,
  ...
);
```

where n is the negative integer that was associated with your BLOB filter when it was declared to the database.

The following declaration defines a table which associates a file name with two user-defined BLOB filters:

```
CREATE TABLE IMAGE_DATA
( IMAGE_ID NUMERIC(18) NOT NULL PRIMARY KEY,
  (FILENAME CHAR(12) NOT NULL,
  BMP BLOB SUB_TYPE -1,
  JPEG BLOB SUB_TYPE -2);
```

Writing an external Blob filter

If you choose to write your own filters, you must have a detailed understanding of the datatypes you plan to transform. Firebird does not do strict datatype checking on BLOB data, but does enforce the rule that BLOB source and target subtypes must be compatible. Maintaining and enforcing this compatibility is the responsibility of your application code.

Filter styles

Filters can be divided into two implementation styles, according to the *timing* of the transformation. If timing is an issue for your application, you should carefully consider which style might better serve your purpose:

- 1 those that convert data one segment at a time. This style reads a segment of data, converts it, and supplies it to the application a segment at a time.
- 2 those that convert data many segments at a time. This style of filter might read all the data and do all the conversion when the BLOB read cursor is first opened, and then simulate supplying data a segment at a time to the application



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Read-only and write-only filters

Some filters may only support reading from or writing to a Blob, but not both operations. If you attempt to use a Blob filter for an operation that it does not support, Firebird returns an error to the application.

Defining the filter function

When writing your filter, you must include an entry point. Firebird calls the filter function when your application performs a BLOB access operation. All communication between Firebird and the filter is through the filter function. The filter function itself may call other functions that comprise the filter executable.

Declare the name of the filter function and the name of the filter executable with the ENTRY_POINT and MODULE_NAME parameters that will appear in the DECLARE FILTER statement of your database. A filter function must have the following declaration calling sequence:

```
filter_function_name(short action, isc_blob_ctl control);
```

The *action* parameter must be one of eight possible action macro definitions

the control parameter, control is an instance of the *isc_blob_ctl* BLOB control structure, defined in the Firebird header file **ibase.h**. These parameters are discussed later in this chapter.

The following listing of a skeleton filter declares the filter function, jpeg_filter:

```
#include <ibase.h>
#define SUCCESS 0
#define FAILURE 1
ISC_STATUS jpeg_filter(short action, isc_blob_ctl control)
{
    ISC_STATUS status = SUCCESS;
    switch (action)
    {
        case isc_blob_filter_open:
        . . .
```



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

```
break;
case isc_blob_filter_get_segment:
. . .
break;
case isc_blob_filter_create:
. . .
break;
case isc_blob_filter_put_segment:
. . .
break;
case isc_blob_filter_close:
. . .
break;
case isc_blob_filter_alloc:
. . .
break;
case isc_blob_filter_free:
. . .
break;
case isc_blob_filter_seek:
. . .
break;
default:
status = isc_uns_ext /* unsupported action value */
. . .
break;
}
return status;
}
```

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

The ellipses (...) in the listing represent code that performs some operations according to the *action* or event that is branched in the case statement.

- Each action is a particular event triggered by some database operation the application might perform.
- Firebird passes two parameters to the filter function:
 - the *action* parameter—one of eight possible actions
 - the *control* parameter, an instance of the *BLOB control structure*, *isc_blob_ctl*

The blob control structure

The BLOB control structure *isc_blob_ctl* provides the essential data exchange between Firebird and the filter.

The *isc_blob_ctl* structure is used in two ways:

- 1 When the application performs a Blob access operation, Firebird calls the filter function and passes it an instance of *isc_blob_ctl*.
- 2 Internal filter functions can pass an instance of *isc_blob_ctl* to internal InterBase access routines.

In either situation, the purpose of certain *isc_blob_ctl* fields depends on the action being performed.

For example, when an application attempts a BLOB INSERT, Firebird passes an

isc_blob_filter_put_segment action to the filter function. The filter function passes an instance of the control structure to Firebird. The *ctl_buffer* of the structure contains the segment data specified by the application to be written via its BLOB INSERT statement.

Because the buffer contains information to pass *into* the filter function, it is called an *IN field*. The filter function should include instructions for performing the write to the database in the *case* branch for the *isc_blob_filter_put_segment*.

For a different phase of processing, for instance when an application attempts a FETCH operation, the *case* branch for the *isc_blob_filter_get_segment* action should include instructions for filling *ctl_buffer* with segment data from the database, to return to the application. Under these conditions, because the buffer is used for filter function output, it is called an *OUT field*.

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

The following table describes each of the fields in the `isc_blob_ctl` Blob control structure, and whether they are used for filter function input (**IN**), or output (**OUT**).

TABLE 26–4 `isc_blob_ctl` structure field descriptions

Field name	Description
<code>*ctl_source</code>	Pointer to the internal InterBase Blob access routine. (IN)
<code>*ctl_source_handle</code>	Pointer to an instance of <code>isc_blob_ctl</code> to be passed to the internal Firebird BLOB access routine. (IN)
<code>ctl_to_sub_type</code>	Target subtype. Information field. Provided to support multi-purpose filters that can perform more than one kind of translation. This field and the next one enable such a filter to decide which translation to perform. (IN)
<code>ctl_from_sub_type</code>	Source subtype. Information field. Provided to support multi-purpose filters that can perform more than one kind of translation. This field and the previous one enable such a filter to decide which translation to perform. (IN)
<code>ctl_buffer_length</code>	For <code>isc_blob_filter_put_segment</code> , field is an IN field that contains the length of the segment data contained in <code>ctl_buffer</code> . For <code>isc_blob_filter_get_segment</code> , field is an IN field set to the size of the buffer pointed to by <code>ctl_buffer</code> , which is used to store the retrieved Blob data.
<code>ctl_segment_length</code>	.Length of the current segment. This field is not used for <code>isc_blob_filter_put_segment</code> . For <code>isc_blob_filter_get_segment</code> , the field is an OUT field set to the size of the retrieved segment (or partial segment, in the case when the buffer length <code>ctl_buffer_length</code> is less than the actual segment length).
<code>ctl_bpb_length</code>	Length of the Blob parameter buffer. Reserved for future enhancement.
<code>*ctl_bpb</code>	Pointer to a Blob parameter buffer. Reserved for future enhancement.

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

TABLE 26-4 isc_blob_ctl structure field descriptions

Field name	Description
*ctl_buffer	Pointer to a segment buffer. For <code>isc_blob_filter_put_segment</code> , field is an IN field that contains the segment data. For <code>isc_blob_filter_get_segment</code> , the field is an OUT field the filter function fills with segment data for return to the application.
ctl_max_segment	Length of longest segment in the Blob. Initial value is 0. The filter function sets this field, for information only.
ctl_number_segments	Total number of segments in the Blob. Initial value is 0. The filter function sets this field, for information only.
ctl_total_length	Total length of the Blob. Initial value is 0. The filter function sets this field, for information only.
*ctl_status	Pointer to the Firebird status vector. (OUT)
ctl_data[8]	8-element array of application-specific data. Use this field to store resource pointers, such as memory pointers and file handles created by the <code>isc_blob_filter_open</code> handler, for example. Then, the next time the filter function is called, the resource pointers will be available for use. (IN/OUT)

Setting control structure information field values

The `isc_blob_ctl` structure contains three fields that store information about the BLOB currently being accessed: `ctl_max_segment`, `ctl_number_segments`, and `ctl_total_length`.

Although Firebird does not use the values of these fields in internal processing, your code should attempt to maintain correct values for these fields in the filter function, whenever possible. Depending on the purpose of the filter, maintaining correct values for the fields is not always possible.

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

For example, a filter that compresses data on a segment-by-segment basis cannot determine the size of `ctl_max_segment` until it processes all segments.

Programming filter function actions

When an application performs a Blob access operation, InterBase passes a corresponding action message to the filter function by way of the action parameter. There are eight possible actions, each of which results from a particular access operation. The following action macro definitions are declared in the `ibase.h` file:

```
#define isc_blob_filter_open 0
#define isc_blob_filter_get_segment 1
#define isc_blob_filter_close 2
#define isc_blob_filter_create 3
#define isc_blob_filter_put_segment 4
#define isc_blob_filter_alloc 5
#define isc_blob_filter_free 6
#define isc_blob_filter_seek 7
```

The following table describes the Blob access operation that corresponds to each action:

TABLE 26-5 Blob access operations

Action	Invoked when ...	Use to ...
<code>isc_blob_filter_open</code>	Application opens a BLOB READ cursor	Set the information fields of the Blob control structure. Perform initialization tasks, such as allocating memory or opening temporary files. Set the status variable, if necessary. The value of the status variable becomes the filter function's return value.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

TABLE 26–5 Blob access operations

Action	Invoked when ...	Use to ...
isc_blob_filter_get_segment	Application executes a BLOB FETCH statement	Set the <code>ctl_buffer</code> and <code>ctl_segment_length</code> fields of the BLOB control structure to contain a segment's worth of translated data on the return of the filter function. Perform the data translation if the filter processes the BLOB segment-by-segment. Set the status variable. The value of the status variable becomes the filter function's return value.
isc_blob_filter_close	Application closes a BLOB cursor	Perform exit tasks, such as freeing allocated memory, closing, or removing temporary files.
isc_blob_filter_create	Application opens a BLOB INSERT cursor	Set the information fields of the Blob control structure. Perform initialization tasks, such as allocating memory or opening temporary files. Set the status variable, if necessary. The value of the status variable becomes the filter function's return value.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

TABLE 26–5 Blob access operations

Action	Invoked when ...	Use to ...
isc_blob_filter_put_segment	Application executes a BLOB INSERT statement	Perform the data translation on the segment data passed in through the BLOB control structure. Write the segment data to the database. If the translation process changes the segment length, the new value must be reflected in the values passed to the writing function. Set the status variable. The value of the status variable becomes the filter function's return value.
isc_blob_filter_alloc	Firebird initializes filter processing; not a result of a particular application action	Set the information fields of the BLOB control structure. Perform initialization tasks, such as allocating memory or opening temporary files. Set the status variable, if necessary. The value of the status variable becomes the filter function's return value.
isc_blob_filter_free	Firebird ends filter processing; not a result of a particular application action	Perform exit tasks, such as freeing allocated memory, closing, or removing temporary files.
isc_blob_filter_seek	Reserved for internal filter use; not used by external filters	



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- ★ Store resource pointers, such as memory pointers and file handles created by the `isc_blob_filter_open` handler, in the `ctl_data` field of the `isc_blob_ctl` Blob control structure. Then, the next time the filter function is called, the resource pointers are still available.

Testing the function return value

The filter function must return an integer indicating the status of the operation it performed. You can have the function return any Firebird status value returned by an internal engine routine.

In some filter applications, a filter function has to supply status values directly. The following table lists status values that apply particularly to BLOB processing:

TABLE 26-6 Blob filter status values

Macro constant	Value	Meaning
SUCCESS	0	Indicates the filter action has been handled successfully. On a Blob read (<code>isc_blob_filter_get_segment</code>) operation, indicates that the entire segment has been read.
FAILURE	1	Indicates an unsuccessful operation. In most cases, a status more specific to the error is returned.
<code>isc_uns_ext</code>	See <code>ibase.h</code>	Indicates that the attempted action is unsupported by the filter. For example, a read-only filter would return <code>isc_uns_ext</code> for an <code>isc_blob_filter_put_segment</code> action.
<code>isc_segment</code>	See <code>ibase.h</code>	During a Blob read operation, indicates that the supplied buffer is not large enough to contain the remaining bytes in the current segment. In this case, only <code>ctl_buffer_length</code> bytes are copied, and the remainder of the segment must be obtained through additional <code>isc_blob_filter_get_segment</code> calls.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

TABLE 26–6 Blob filter status values

Macro constant	Value	Meaning
isc_segstr_eof	See ibase.h	During a Blob read operation, indicates that the end of the Blob has been reached; there are no additional segments remaining to be read.

For more information about Firebird status values, see the *Firebird Reference Guide*.

BLOB descriptors

A BLOB descriptor is used to provide dynamic access to BLOB information. For example, it can be used to store information about BLOB data for filtering (conversion) purposes, such as character set information for text Blob data and subtype information for text and non-text BLOB data. Two BLOB descriptors are needed whenever a filter will be used when writing to or reading from a BLOB: one to describe the filter source data, and the other to describe the target.

A BLOB descriptor is a structure defined in the ibase.h header file as follows:

```
typedef struct {
    short blob_desc_subtype; /* type of Blob data */
    short blob_desc_charset; /* character set */
    short blob_desc_segment_size; /* segment size */
    unsigned char blob_desc_field_name [32]; /* Blob column name */
    unsigned char blob_desc_relation_name [32]; /* table name */
} ISC_Blob_DESC;
```

 For more information about the character sets recognized by Firebird, see the *Firebird Reference Guide*.

The segment size of a BLOB is the maximum number of bytes that an application is expected to write to or read from the BLOB. You can use this size to allocate your own buffers.

The `blob_desc_relation_name` and `blob_desc_field_name` fields contain null-terminated strings.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Ways to populate a BLOB descriptor

There are four possible ways to populate a BLOB descriptor. You can do so by:

- ◆ Calling `isc_blob_default_desc()`. This stores default values into the descriptor fields. The default subtype is 1 (TEXT), segment size is 80 bytes, and charset is the default charset for your process.
- ◆ Calling `isc_blob_lookup_desc()`. This accesses the database system metadata tables to look up and copy information for the specified Blob column into the descriptor fields.
- ◆ Calling `isc_blob_set_desc()`. This initializes the descriptor from parameters you call it with, rather than accessing the database metadata.
- ◆ Setting the descriptor fields directly.

The following example calls `isc_blob_lookup_desc()` to look up the current subtype and character set information for a BLOB column named PROJ_DESC in a table named PROJECT. It stores the information into the source descriptor, `from_desc`.

```
isc_blob_lookup_desc (
    status_vector,
    &db_handle; /* Set by previous isc_attach_database() call. */
    &tr_handle, /* Set by previous isc_start_transaction() call. */
    "PROJECT", /* Table name. */
    "PROJ_DESC", /* Column name. */
    &from_desc, /* Blob descriptor filled in by this function call. */
    &global /* Global column name, returned by this function. */
)
```



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Requesting filtering in an application

To request filtering of Blob data as it is read from or written to a Blob, follow these steps in your application:

- 1 Create a BLOB parameter buffer (BPB) specifying the source and target subtypes, and optionally character sets (for TEXT subtypes).
- 2 Call either `isc_open_blob2()` or `isc_create_blob2()` to open a Blob for read or write access, respectively. In the call, pass the BPB, whose information Firebird will use to determine which filter should be called.

The BLOB parameter buffer

A BLOB parameter buffer (BPB) is needed whenever a filter will be used when writing to or reading from a Blob. The BPB is a char array variable, specifically declared in an application, that contains the source and target subtypes. When data is read from or written to the Blob associated with the BPB, Firebird will automatically invoke the filter appropriate to the source and target subtypes specified in the BPB.

If the source and target subtypes are both 1 (TEXT), and the BPB also specifies different source and target character sets, then when data is read from or written to the Blob associated with the BPB, Firebird will automatically convert each character from the source to the target character set.

A Blob parameter buffer can be generated in one of two ways:

1. Indirectly, through API calls to create source and target descriptors and then generate the BPB from the information in the descriptors.
2. Directly by populating the BPB array with appropriate values.

If you generate a BPB via API calls, you do not need to know the format of the BPB. But if you wish to directly generate a BPB, then you must know the format.

Both approaches are described in the following sections. The format of the BPB is documented in the section about directly populating the BPB.

Using API calls to generate a BLOB parameter buffer

To generate a BPB indirectly, use API calls to create source and target BLOB descriptors, and then call `isc_blob_gen_bpbb()` to generate the BPB from the information in the descriptors. Follow these steps:

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

- 1 Declare two Blob descriptors, one for the source, and the other for the target.

For example,

```
#include <ibase.h>
ISC_Blob_DESC from_desc, to_desc;
```

- 2 Store appropriate information in the Blob descriptors, by either

- calling one of the functions `isc_blob_default_desc()`, `isc_blob_lookup_desc()`, or `isc_blob_set_desc()`

or

- setting the descriptor fields directly.

The following example looks up the current subtype and character set information for a Blob column named GUIDEBOOK in a table named TOURISM, and stores it into the source descriptor, `from_desc`. It then sets the target descriptor, `to_desc` to the default subtype (TEXT) and character set, so that the source data will be converted to plain text.

```
isc_blob_lookup_desc (
status_vector,
&db_handle; /* set in previous isc_attach_database() call */
&tr_handle, /* set in previous isc_start_transaction() call */
"TOURISM", /* table name */
"GUIDEBOOK", /* column name */
&from_desc, /* Blob descriptor filled in by this function call */
&global);
if (status_vector[0] == 1 && status_vector[1])
{
/* process error */
isc_print_status(status_vector);
return(1);
};
```



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

```
isc_blob_default_desc (
    &to_desc, /* Blob descriptor filled in by this function call */
    "", /* NULL table name; it's not needed in this case */
    ""); /* NULL column name; it's not needed in this case */
```

- 3 Declare a character array which will be used as the BPB. Make sure it is at least as large as all the information that will be stored in the buffer.

```
char bpb[ 20 ];
```

- 4 Declare an unsigned short variable into which Firebird will store the actual length of the BPB data:

```
unsigned short actual_bpb_length;
```

- 5 Call `isc_blob_gen_bpб()` to populate the BPB based on the source and target BLOB descriptors passed to `isc_blob_gen_bpб()`.

For example,

```
isc_blob_gen_bpб(
    status_vector,
    &to_desc, /* target Blob descriptor */
    &from_desc, /* source Blob descriptor */
    sizeof(bpb), /* length of BPB buffer */
    bpb, /* buffer into which the generated BPB will be stored
    */
    &actual_bpb_length /* actual length of generated BPB */
);
```

Generating a BPB directly

It is possible to generate a BPB directly. It consists of the following parts:

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

- 1 A byte specifying the version of the parameter buffer, always the compile-time constant, `isc_bpbb_version1`.
- 2 A contiguous series of one or more clusters of bytes, each describing a single parameter.

Each cluster consists of the following parts:

- 1 A one-byte parameter type. There are compile-time constants defined for all the parameter types (for example, `isc_bpbb_target_type`).
- 2 A one-byte number specifying the number of bytes that follow in the remainder of the cluster.
- 3 A variable number of bytes, whose interpretation depends on the parameter type.

The BPB must contain `isc_bpbb_version1` at the beginning, and must contain clusters specifying the source and target subtypes. Character set clusters are optional. If the source and target subtypes are both 1 (TEXT), and the BPB also specifies different source and target character sets, then when data is read from or written to the BLOB associated with the BPB, Firebird will automatically convert each character from the source to the target character set.

The following is an example of directly creating a BPB for a filter whose source subtype is -4 and target subtype is 1 (TEXT):

```
char bpbb[] = {  
    isc_bpbb_version1,  
    isc_bpbb_target_type,  
    1, /* # bytes that follow which specify target subtype */  
    1, /* target subtype (TEXT) */  
    isc_bpbb_source_type,  
    1, /* # bytes that follow which specify source subtype */  
    -4, /* source subtype*/  
};
```

If you do not know the source and target subtypes until run time, you provide for run-time assignments to those values in the appropriate BPB locations.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Expressing numbers in the BPB

All numbers in the BLOB parameter buffer must be represented in a *generic format*, with the least significant byte first, and the most significant byte last. Signed numbers should have the sign in the last byte. The API function `isc_portable_integer()` can be used to reverse the byte order of a number.

The following table lists the parameter types and their meaning:

TABLE 26-7 Blob parameter buffer parameter types

Parameter type	Description
<code>isc_bpb_target_type</code>	Target subtype
<code>isc_bpb_source_type</code>	Source subtype
<code>isc_bpb_target_interp</code>	Target character set
<code>isc_bpb_source_interp</code>	Source character set

Making the filter request

The request to use a filter is made when a BLOB is created or opened for read or write access. In the call to `isc_create_blob2()` or `isc_open_blob2()`, pass the BPB, which Firebird will read to determine which filter should be called.

The following example illustrates creating and opening a Blob for write access.

```
isc_blob_handle blob_handle; /* declare at beginning */
ISC_QUAD blob_id; /* declare at beginning */
...
isc_create_blob2(
    status_vector,
    &db_handle,
```



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

```
&tr_handle,  
&blob_handle, /* to be filled in by this function */  
&blob_id, /* to be filled in by this function */  
actual_bpblength, /* length of BPB data */  
&bpb /* Blob parameter buffer */  
)  
if (status_vector[0] == 1 && status_vector[1])  
{  
isc_print_status(status_vector);  
return(1);  
}
```



For further information about writing data to a BLOB and updating a BLOB column of a table row to refer to the new BLOB, see Chapter 7, *Working with Blob Data*, in the API Guide (APIGuide.pdf) of the InterBase® 6 documentation set, obtainable from Borland.

From the same source, further details are available regarding additional steps required for selecting and opening a BLOB for read access.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

CHAPTER 27

Migrating to Firebird

If you are migrating a pre-version 6 InterBase® database to Firebird, a number of changes and new features will affect the behavior of your database. With an InterBase® 6.0 database, new features are of less concern to you than some bugs and holes that have been closed up during Firebird development. Most of those issues affect all versions of InterBase® equally.

The four Ds

InterBase® to Firebird migration issues can be summarized into four broad categories, all starting with the letter 'D'. They are

Dialects Borland introduced the concept of *dialects* into Firebird's ancestor, InterBase®, during the design and development of version 6, whose open sourced code became the codebase for Firebird 1. Dialects act as a shielding and transition mechanism to enable older databases to be used with the Firebird server and client whilst easing into the new features of Firebird.

Data types Date and time data types all change in Dialect 3, as do numeric types and the way that they are stored and calculated.

Double-quoted SQL identifiers The SQL standard provides for using previously "illegal" strings—notably reserved words and words containing blank spaces—as object identifiers, as long as they are defined and always used with the double-quote symbol delimiting them. If used, this feature will affect existing client application code and the scripts defining your stored procedures and triggers.

Defence Firebird closed up two major sources of trouble: the potential for anomalous connection path strings for clients to a Windows server, which corrupted databases irrevocably; and the SQL parser's acceptance of ambiguous correlations in statements, which could cause views and data sets to output wrong data. These fixes will cause errors on Firebird until the effects are dealt with on the migrating databases.



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

The migration checklists

Moving your databases affects the databases themselves and also, possibly, your client application software. The following list summarizes the migration tasks:

Database migration

- 1 Back up all databases that are for migration—using *transportable* option
- 2 Disable or uninstall InterBase® on the servers
- 3 Install the Firebird server
- 4 Restore databases to be migrated
- 5 Validate migrated databases
- 6 Transform databases to SQL dialect 3

Client migration

- 1 Identify the client machines and applications that must be upgraded to Firebird
- 2 Identify areas in your application that might need upgrading
- 3 Remove or rename old versions of the client library
- 4 Install the Firebird client library on each machine that requires it
- 5 Upgrade SQL applications to SQL dialect 3

Migration considerations

In Firebird, each client and database has an *SQL dialect attribute*—an indicator that instructs a Firebird server how to interpret *transition features*: those features whose meanings have changed between versions.

Dialects

- Dialect 1: Firebird interprets transition features as an InterBase® 5 server would

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

- Dialect 2: Firebird recognizes transition features and flags them with a warning. Only a client can carry the Dialect 2 attribute.
- Dialect 3: Firebird interprets transition features as Firebird and SQL-92 compliant

Transition features

- Double quote (") : changed from a synonym for the single quote ('') to the delimiter for an object name
- Large exact numerics: DECIMAL and NUMERIC datatypes with precision greater than 9 are stored as INT64 (NUMERIC(18)) instead of DOUBLE PRECISION
- DATE, TIME, and TIMESTAMP datatypes:
 - DATE has changed from a 64-bit quantity containing both date and time information to a 32-bit quantity containing only date information
 - TIME is a 32-bit quantity containing only time information
 - TIMESTAMP is a 64-bit quantity containing both date and time information (same as DATE in the SQL supported by InterBase® 5 and earlier)

Defence features

- Connection path strings: The first user to log into a Firebird database opens it in exclusive access mode. This blocks only applications and BDE aliases that connect with a path string that is different (from the point of view of the file system) to that which was used for the first connection. See The connection path bug.
- Ambiguous correlations of columns in statements involving multiple tables (joins and subqueries) will raise an exception on a Firebird server, whether they occur in a client query or in compiled database objects.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

On-disk structure

The change from InterBase® version 5 to version 6 or Firebird entailed a major change in on-disk structure (ODS). Table 27–1 summarizes the compatibility issues you face with different combinations of SQL dialect, client version, server version, and ODS. ODS 8 is InterBase® 4, ODS 9 is InterBase® 5 and ODS 10 is Firebird and InterBase® 6. Shaded rows show combinations that do not work.

TABLE 27–1 Function by SQL dialect, client version, server version, and ODS level

Dialect	Version			Server functionality
	Client	Server	ODS	
1	IB5 or Firebird	IB5	8 or 9	InterBase 5 functionality.
			10	Server cannot open database.
		Firebird	8, 9, or 10	Fields defined using dialect 1 behave in the old way. Columns defined using 64-bit data types cannot be accessed. The least significant 32 bits of a generator are returned as an integer. Computations yield the results and result types of InterBase 5.
2	Firebird	IB5	8 or 9	Error: server does not understand the dialect.
		IB5	10	Server cannot open database.
	Firebird	Firebird	8, 9, or 10	Data access for columns created with InterBase® 5 is the same as in InterBase® 5; for columns created with Firebird 1, data access has Firebird 1 functionality. All defines involving numeric or decimal data types produce Firebird 1 functionality, but also generate a warning message, because behavior differs between InterBase® 5 and Firebird 1.

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**TABLE 27-1 Function by SQL dialect, client version, server version, and ODS level (*continued*)

Dia- lect	Version			Server functionality
	Client	Server	ODS	
3	Firebird	5	8 or 9	Error: server does not understand the dialect.
		5	10	Server cannot open database.
	Firebird	8, 9, or 10		Data access for columns created with InterBase 5 is the same as in InterBase 5; for columns created with Firebird 1, data access has full Firebird 1 functionality. All defines involving numeric or decimal data use integral types.

Servers and databases

Databases created using Firebird 1 are stored in ODS 10 data format. This format is not backward compatible with older formats.

- If you upgrade a server to Firebird 1, you should *eventually* migrate the databases that it accesses to Firebird 1 as well. Firebird 1 servers can read from and write to InterBase 5 databases. Databases earlier than 5 must be upgraded to Firebird.
- If you do not upgrade a server, do not migrate the databases that it accesses. InterBase® 5 and earlier servers cannot access Firebird 1 databases.

Clients and databases

Applications using InterBase 5® and written to connect through an InterBase® client work with the Firebird 1 server and its databases, with some restrictions:

- InterBase® 5 clients cannot access dialect 3 columns stored as INT64, TIME, or DATE.
- InterBase® 5 clients cannot display new datatypes in metadata using the SHOW command or any equivalent.
- InterBase 5® clients interpret the DATE datatype as TIMESTAMP, because that was the definition of DATE prior to Firebird 1.

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

- InterBase® 5 clients cannot access any object named with a delimited identifier.
- Clients that use the Borland Database Engine (BDE) at a lower level than v.5.2 to access a Firebird 1 server are not able to access any of the new datatypes, regardless of the InterBase client version.

Reserved words used as identifiers

If you migrate to Firebird dialect 1 an older database that uses any of the newer InterBase® 6 or Firebird 1 reserved words as identifiers, access to the objects that bear those reserved word identifiers varies with client version and dialect:

- InterBase 5 clients can still access the objects
- Firebird 1, dialect 1 clients *cannot* access the objects
- Firebird 1, dialect 3 clients can access the objects only by enclosing the object identifiers in double quotes.
- If you are migrating an InterBase® 6 database, be on your guard for new keywords added to the reserved list by Firebird and InterBase® 6.5. These are included in the *Firebird Reference Guide* and listed in the release notes for Firebird 1.

For example, the following statement uses the new reserved word TIME as a column identifier:

```
SELECT TIME FROM atable;
```

When an InterBase 5 client executes this statement, a Firebird 1 server returns the same information as did previous versions. A Firebird 1 server permits an InterBase 5 client to use the new reserved words as object identifiers; the server recognizes that these clients were created at a time when such identifiers were not reserved words.

When a Firebird 1 client executes this statement, a Firebird 1 server returns an error, because TIME is now a reserved word. A Firebird 1 client would have to express the statement like this:

```
SELECT "TIME" FROM atable;
```



See [New reserved words](#) for a list of reserved words new in InterBase® 6 and in Firebird 1. The full list is in *Firebird Reference Guide*— [Reserved Words](#) (ch. 7 p. 276).



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

SQL dialects

Clients and databases have dialects. A servers does not have a dialect; it interprets data structures and client requests according to the dialect attributes of each database and client connection.

Features available in all dialects

The following Firebird 1 features are available in both dialect 1 and dialect 3:

- The ALTER COLUMN clause of the ALTER TABLE statement
- Altering domain definitions with ALTER DOMAIN
- The TIMESTAMP datatype, which is the equivalent of the DATE datatype in InterBase® pre v.6
- The EXTRACT() function
- Context variables CURRENT_TIMESTAMP, CURRENT_USER and CURRENT_ROLE
- Read-only databases
- SQL warnings
- The Services API (on SuperServer only) and Install API

Features available only in dialect 3

The following features conflict with dialect 1 usage, and so are available only in dialect 3 clients and databases. Previous versions of InterBase allowed the use of both single and double quotes around string literals. In Firebird 1, double quotes are used only for delimited identifiers, and strings must be enclosed in single quotes.

- Delimited identifiers

If you enclose an identifier in double quotes it

- can mimic a reserved word
- can contain spaces
- must be case sensitive
- can contain diacritic ASCII characters

An identifier with any of these characteristics *must* be delimited by *double* quotes. Text delimited by single quotes is interpreted as a string constant.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

❖ For simplicity of maintenance and clarity of code, it is preferable to avoid delimited identifiers unless you have a prevailing reason to use them. If you intend to use a tool for converting or recreating your database and your existing database would not break any rules with respect to identifiers, then choose a tool which makes delimited identifiers optional.

- INT64 data storage

In dialect 3, data in DECIMAL and NUMERIC columns are stored as INT64 (NUMERIC(18)) when the precision is greater than 9. This is true only for columns that are *created* in dialect 3. These same datatypes are stored as DOUBLE PRECISION in dialect 1 and in all InterBase database versions earlier than Firebird 1.

❖ This change in numeric storage also requires different arithmetic algorithms. For an explanation, see [Using exact numeric data types in arithmetic](#) on p. 238 of chapter 13.

- DATE and TIME datatypes

In dialect 3, the DATE datatype holds only date information, and the new TIME datatype holds only time information.

Dialect 1 clients and databases

In dialect 1, a Firebird 1 server interprets transition features just as InterBase 5 servers did:

- Double-quoted text is interpreted as a string literal. Delimited identifiers are not available.
- The DATE datatype contains time information as well as date information, and is interpreted as TIMESTAMP. Dialect 1 clients expect the entire timestamp to be returned. In dialect 1, DATE and TIMESTAMP are identical.
- The TIME datatype is not available.
- Dialect 1 databases store DECIMAL and NUMERIC datatypes with precision greater than 9 as DOUBLE PRECISION, not INT64. Dialect 1 clients expect information stored in these datatypes to be returned as double precision; such clients cannot create database fields to hold 64-bit integers.

A Firebird 1 server recognizes all other (non-transition) features in dialect 1 clients and databases.

Firebird 1 servers can send warnings to Firebird 1 dialect 1 clients, but not to InterBase 5 or earlier clients.



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

Dialect 2 clients

Dialect 2 is available only on the client side. It is intended for assessing possible problems in legacy metadata that is being migrated to dialect 3. To determine where the problem spots are when you migrate a database from dialect 1 to dialect 3, you extract the metadata from the database, set **isql** to dialect 2, and then run that metadata file through **isql** as a script. **isql** issues a warning whenever it encounters double quotes, DATE datatypes, or large exact numerics, to alert you to places where you might need to change the metadata in order to make a successful migration to dialect 3.

To detect problem areas in the metadata of a database that you want to transform, extract the metadata and run it through a dialect 2 client, which will report all instances of transition features. For example:

```
isql -i v5metadata.sql
```

Do not try to set databases to Dialect 2.

Dialect 3 clients and databases

In dialect 3, a Firebird 1 server interprets transition features as Firebird 1 and SQL-92 compliant:

- Double-quoted strings are treated as delimited identifiers.
- Dialect 3 DATE datatype fields contain only date information. Dialect 3 clients expect only date information from a field of datatype DATE.
- The TIME datatype is available, and stores only time information.
- Dialect 3 databases store DECIMAL and NUMERIC datatypes with precision greater than 9 as INT64 (NUMERIC(18)), *if and only if they are in columns that were created in dialect 3*. Dialect 3 clients expect DECIMAL and NUMERIC datatypes with precision greater than 9 to be returned as INT64. (To learn how to migrate older data to INT64 storage, see [Transforming databases to Dialect 3](#) and [Converting NUMERIC and DECIMAL datatypes](#).)

A Firebird 1 server recognizes all other non-transition features in dialect 3 clients and databases.

Setting SQL dialect

You can specify dialect with **isql**, **gpre**, or **gfix**.



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

Setting **isql** client dialect

To use **isql** to create a database in a particular dialect, first set **isql** to the desired dialect and then create the database. You can set **isql** dialect the following ways.

- On the command line, start **isql** with option **-sql_dialect n**, where **n** is 1, 2, or 3:

```
isql -sql_dialect n
```

- Within an **isql** session or in a SQL script, include the following statement:

```
SET SQL DIALECT n;
```

isql dialect precedence is as follows:

Lowest: Dialect of an attached Firebird 1 database

Next lowest: Dialect specified on the command line

Next highest: Dialect specified during the session

Highest: Dialect of an attached InterBase 5 database (= 1)

In Firebird 1, **isql** has the following behavior with respect to dialects:

- If you start **isql** and attach to a database without specifying a dialect, **isql** takes on the dialect of the database.
 - If you specify a dialect on the command line when you invoke **isql**, it retains that dialect after connection unless explicitly changed.
 - When you change the dialect during a session using [SET SQL DIALECT n](#), **isql** continues to operate in that dialect until it is explicitly changed.
 - When you create a database using **isql**, the database is created with the dialect of the **isql** client; for example, if **isql** has been set to dialect 1, when you create a database, it is a dialect 1 database.
 - If you create a database without first specifying a dialect for the **isql** client or attaching to a database, **isql** creates the database in dialect 3.
- ❖ Any Firebird 1 **isql** client that attaches to an InterBase 5 database resets to dialect 1.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Setting gpre client dialect

In Firebird 1, by default **gpre** takes on the dialect of the database to which it is connected. This enables **gpre** to parse pre-Firebird source files without modification.

You can set **gpre** to operate as a client in a different dialect these ways:

- Start **gpre** with option **-sql_dialect** *n*, where *n* is 1, 2, or 3:

```
gpre -sql_dialect n
```

- Specify dialect within the source; for example:

```
EXEC SQL  
SET SQL DIALECT n
```

gpre dialect precedence is as follows:

Lowest: Dialect of an attached database

Middle: Dialect specified on the command line

Highest: Dialect specified in the source

Setting gfix database dialect

The command-line option **-sql_dialect** *n*, where *n* is 1 or 3, sets the dialect of an ODS 10 database. For example, the following statement sets **mydb.gdb** to dialect 3:



Symbols **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

```
gfix -sql_dialect 3 mydb.gdb
```

New reserved words

InterBase® 6.0 introduced the following reserved words, which are also reserved words in Firebird 1.0:

COLUMN	DAY	MONTH	TYPE
CURRENT_DATE	EXTRACT	SECOND	WEEKDAY
CURRENT_TIME	HOUR	TIME	YEAR
CURRENT_TIMESTAMP	MINUTE	TIMESTAMP	YEARDAY

Firebird 1 introduced the following reserved words:

BREAK	CURRENT_ROLE	CURRENT_USER	DESCRIPTOR
FIRST	RECREATE	SKIP	SUBSTRING

The following words are not yet reserved, but should be avoided because they will be implemented later:

ABS	BOTH	CASE	CHAR_LENGTH
CHARACTER_LENGTH	COALESCE	IIF	LEADING
NULLIF	OCTET_LENGTH	TRIM	TRAILING

The following new keywords were introduced in InterBase® 6.5:

PERCENT	ROWS	TIES
---------	------	------

Reserved word restrictions

- You cannot create objects in a Firebird 1 dialect 1 database that have any of these reserved words as object names (identifiers).
- You can migrate an InterBase 5 database that contains these reserved words used as identifiers to Firebird 1 dialect 1 without changing the object names: a column could be named "YEAR", for instance.
- InterBase 5 clients can access these reserved word identifiers without error.

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

- Firebird 1 clients *cannot* access reserved words that are used as identifiers. In a dialect 1 database, you must change the names so that they are not reserved words.
- If you migrate directly to dialect 3, you can retain the names, but you must delimit them with double quotes. To retain accessibility for older clients, put the names in all upper case. Delimited identifiers are case sensitive.
- Although TIME is a reserved word in Firebird 1 dialect 1, you cannot use it as a datatype because such databases guarantee datatype compatibility with InterBase 5 clients.
- In dialect 3 databases and clients, any reserved word can be used as an identifier as long as it is delimited with double quotes.

The double-quotes issue

Up to and including InterBase® 5, the use of either single or double quotes around string constants was permitted. The concept of delimited identifiers did not exist. Firebird 1 supports delimited identifiers, which means that double quotes must be used *only* for delimited identifiers.

Single quotes and double quotes: summary

- In all versions of InterBase, anything delimited by single quotes is treated as a string constant.
 - In InterBase® 5 and older, string constants could be delimited by either double or single quotes. Firebird 1 dialect 1 is a transition mode that behaves like older versions of InterBase with respect to quote marks: it interprets strings within double quotes as string constants and does not permit delimited identifiers.
 - Firebird 1 dialect 3 uses double quotes only for delimited identifiers. String constants must be delimited by single quotes, never double.
 - See Table 27–2 for how to handle quotations marks that occur *within* strings or identifiers.
- ❖ Transition features should be considered as *deprecated*. In a future release, Firebird may disallow the dialect 1 behaviors in all incoming databases.

DDL and DML statements

When a Firebird 1 server detects that a client is dialect 1, the server permits client DML statements to contain double quotes, correctly interpreting anything delimited by double quotes as a string constant. However, the



Symbols **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

server does not permit double quotes in client DDL statements, because such metadata would not be allowed in dialect 3. Firebird 1 servers insist that string constants be delimited with single quotes when clients create new metadata.

DATE, TIME, and TIMESTAMP datatypes

Firebird 1 dialect 3 replaces the old InterBase DATE datatype, which contains both date and time information, with SQL 92 standard TIMESTAMP, DATE, and TIME datatypes. The primary migration problem exists in the source code of application programs that use the InterBase 5 DATE datatype. In Firebird 1, the DATE reserved word represents a date only datatype, while in InterBase 5 DATE represents a date and time datatype.

Columns and domains that are defined as DATE datatype in InterBase 5 DATE appear as TIMESTAMP columns when the database is restored in Firebird 1. However, a TIMESTAMP datatype has four decimal points of precision, while an InterBase 5 DATE datatype has only two decimal points of precision.

If you migrate your database to dialect 3 and you require only date or only time information from a TIMESTAMP column, you can use ALTER COLUMN to change the datatype to DATE or TIME. These columns each take only four bytes, whereas TIMESTAMP and the InterBase 5 DATE columns each take eight bytes. If your TIMESTAMP column holds both date and time information, you cannot change it to an Firebird 1 DATE or TIME column using ALTER COLUMN, because ALTER COLUMN does not permit data loss.

- In dialect 1, only TIMESTAMP is available. TIMESTAMP is the equivalent of the DATE datatype in previous versions. When you back up an older database and restore it in Firebird 1, all the DATE columns and domains are automatically restored as TIMESTAMP. DATE and TIMESTAMP datatypes are both available and both mean the same thing in dialect 1.
- In dialect 3, TIMESTAMP functions as in dialect 1, but two additional datatypes are available: DATE and TIME. These datatypes function as their names suggest: DATE holds only date information and TIME holds only time.
- In dialect 3, DATE and TIME columns require only four bytes of storage, while TIMESTAMP columns require eight bytes.

The following example show the differences between dialect 1 and dialect 3 clients when date information is involved.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

```
CREATE TABLE table1 (fld1 DATE, fld2 TIME);
INSERT INTO table1 VALUES (CURRENT_DATE, CURRENT_TIME);
```

Using dialect 1 clients

```
SELECT * FROM table1;
Statement failed, SQLCODE = -804
Dynamic SQL Error
-SQL error code = -804
-Data type unknown
-Client SQL dialect 1 does not support reference to TIME datatype
```

```
SELECT fld1 FROM table1;
Statement failed, SQLCODE = -206
Dynamic SQL Error
-SQL error code = -206
-Column unknown
-FLD1
-Client SQL dialect 1 does not support reference to DATE datatype
```



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

Using dialect 3 clients

```
SELECT * FROM table1;
FLD1          FLD2
=====
1999-06-25  11:32:30.0000
SELECT fld1 FROM table1;
FLD1
=====
1999-06-25
CREATE TABLE table1 (fld1 TIMESTAMP);
INSERT INTO table1 (fld1) VALUES (CURRENT_TIMESTAMP);
SELECT * FROM table1;
```

In dialect 1:

```
FLD1
=====
25-JUN-1999
```

In dialect 3:

```
FLD1
=====
1999-06-25 10:24:35.0000
```

```
SELECT CAST (fld1 AS CHAR(5)) FROM table1;
```

In dialect 1:

```
=====
25-JU
```



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

In dialect 3:

```
Statement failed, SQLCODE = -802
arithmetic exception, numeric overflow, or string truncation
```

EXTRACT()

The EXTRACT operator allows you to return different parts of a TIMESTAMP value. The EXTRACT operator makes no distinction between dialects when formatting or returning the information.

```
SELECT EXTRACT (YEAR FROM timestamp_fld) FROM table_name;
=====
1999
SELECT EXTRACT (YEAR FROM timestamp_fld) FROM table_name;
=====
1999
SELECT EXTRACT (MONTH FROM timestamp_fld) FROM table_name;
=====
6
SELECT EXTRACT (DAY FROM timestamp_fld) FROM table_name;
=====
25
SELECT EXTRACT (MINUTE FROM timestamp_fld) FROM table_name;
=====
24
SELECT EXTRACT (SECOND FROM timestamp_fld) FROM table_name;
=====
35.0000
SELECT EXTRACT (WEEKDAY FROM timestamp_fld) FROM table_name;
=====
5
```

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

```
SELECT EXTRACT (YEARDAY FROM timestamp_fld) FROM table_name;
```

```
=====
```

```
175
```

```
SELECT EXTRACT (MONTH FROM timestamp_fld) ||  
'-' || EXTRACT (DAY FROM timestamp_fld) ||  
'-' || EXTRACT (YEAR FROM timestamp_fld) FROM table_name;
```

```
=====
```

```
6-25-1999
```

Converting TIMESTAMP columns to DATE or TIME

If you transform a database to dialect 3, any columns that previously had the DATE datatype would have the TIMESTAMP datatype. If you want to store that data in a DATE or TIME column, follow these steps:

- 1 Use ALTER TABLE to create a new column of the desired type.
- 2 Insert the values from the original column into the new column:

```
UPDATE tablename SET new_field = CAST (old_field AS new_field);
```

- 3 Use ALTER TABLE to drop the original column.
- 4 Use ALTER TABLE ... ALTER COLUMN to rename the new column.

Casting date/time datatypes

Firebird 1 dialect 3 no longer allows the use of the CAST operator to remove the date portion of a timestamp by casting the timestamp value to a character value. When you cast a TIMESTAMP to a CHAR or VARCHAR in dialect 3, the destination type must be at least 24 characters in length or InterBase will report a string overflow exception. This is required by the SQL-99 standard.

Casting DATE to string results in YYYY-MM-DD where "MM" is a two-digit month. If the result does not fit in the string variable a string truncation exception is raised. In earlier versions, this case resulted in DD-Mon-YYYY HH:mm:SS.hundreds where "Mon" was a three-letter English month abbreviation. Inability to fit in the string variable resulted in a silent truncation.

Casting a string to a date now permits strings of the form:

```
'YYYY-mm-dd'      'YYYY/mm/dd'      'YYYY mm dd'  
'YYYY:mm:dd'     'YYYY.mm.dd'
```



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

In all of the forms above, you can substitute a month name or three-letter abbreviation in English for the two-digit numeric month. However, the order must always be four-digit year, then month, then day.

In previous versions of InterBase, you could enter date strings in a number of forms, including forms that had only two digits for the year. Those forms are still available in Firebird 1. If you enter a date with only two digits for the year, InterBase uses its "sliding window" algorithm to assign a century to the years.

In Firebird 1, you can cast a string to a date earlier than the year 100:

```
EXTRACT(YEAR FROM CAST('31-Dec-0001' AS DATE))
```

sets the value of YEAR to 1. Previously, InterBase ignored leading zeros, so 0001 was equivalent to 01, which would be interpreted as the year 2001.

For more information about casting date and time datatypes., see [Casting from date and time datatypes to other SQL datatypes](#) on p. 256 of chapter 13.

DECIMAL and NUMERIC datatypes

The following sections highlight some of the changes introduced by Firebird 1 when dealing with numeric values. They need to be considered carefully when migrating your database from dialect 1 to dialect 3. When considering these issues, keep in mind that in order to make use of the new functionality, the statements must be created with a client dialect setting of 3.

The most notable problems when migrating to Firebird 1 involve using the division operator and the AVG() function (which also implies division) with exact numeric operands. *Exact numeric* refers to any of the following data types: INTEGER, SMALLINT, DECIMAL, or NUMERIC. NUMERIC and DECIMAL datatypes that have a precision greater than 9 are called *large exact numerics*. Large exact numerics are stored as DOUBLE PRECISION in dialect 1 and as INT64 in columns created in dialect 3.

- ❖ When you migrate an exact numeric column to dialect 3 it is still stored as DOUBLE PRECISION.

The migration does not change the way the data is stored because INT64 cannot store the whole range that DOUBLE PRECISION can store. There is potential data loss, so InterBase does not permit direct conversion. If you decide that you want your data stored as INT64, you must create a new column and copy the data. Only exact numeric columns that are *created* in dialect 3 are stored as INT64. The details of the process are provided in [Transforming databases to Dialect 3](#).

You might or might not want to change exact numeric columns to INT64 when you migrate to dialect 3—see [Do you really need to migrate your NUMERIC and DECIMAL datatypes?](#) on page 647 for a discussion of issues.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Dialect 3 features and changes include

- Support for 64 bit integers.
- Overflow protection. In dialect 1, if the product of two integers was bigger than 31 bits, the product was returned modulo 2^{32} . In dialect 3, the true result is returned as a 64-bit integer. Further, if the product, sum, difference, or quotient of two exact numeric values is bigger than 63 bits, InterBase issues an arithmetic overflow error message and terminates the operation. (Previous versions sometimes returned the least-significant portion of the true result.). The stored procedure **bignum** below demonstrates this.

Operations involving division return an exact numeric if both operands are exact numerics in dialect 3. When the same operation is performed in dialect 1, the result is a DOUBLE PRECISION.

To obtain a DOUBLE PRECISION quotient of two exact numeric operands in dialect 3, explicitly cast one of the operands to DOUBLE PRECISION before performing the division:

```
CREATE TABLE table 1 (n1 INTEGER, n2 INTEGER);
INSERT INTO table 1 (n1, n2) VALUES (2, 3);
SELECT n1 / n2 FROM table1;
=====
0
```

Similarly, to obtain a double precision value when averaging an exact numeric column, you must cast the argument to double precision before the average is calculated:

```
SELECT AVG(CAST(int_col AS DOUBLE PRECISION))FROM table1;
```



For more details, see [Operations using exact numerics](#) on p. 239 of chapter 13

Compiled objects

The behavior of a compiled object such as a stored procedure, trigger, check constraint, or default value depends on the dialect setting of the client at the time the object is compiled. Once compiled and validated by the server the object is stored as part of the database and its behavior is constant regardless of the dialect of the client that calls it.

Consider the following procedure:

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

```
CREATE PROCEDURE exact1 (a INTEGER, b INTEGER) RETURNS (c INTEGER)
AS BEGIN
  c = a / b;
  EXIT;
END;
```

When created by a dialect 1 client:

EXECUTE PROCEDURE *exact 1* returns 1 when executed by either a dialect 1 or dialect 3 client.

When created by a dialect 3 client:

EXECUTE PROCEDURE *exact 1* returns 0 when executed by either a dialect 1 or dialect 3 client.

Consider the following procedure:

```
CREATE PROCEDURE bignum (a INTEGER, b INTEGER) RETURNS (c NUMERIC(18,0))
AS BEGIN
  c = a * b;
  EXIT;
END;
```

When created by a dialect 1 client:

EXECUTE PROCEDURE *bignum* (65535, 65535) returns -131071.0000 when executed by either a dialect 1 or dialect 3 client.

When created by a dialect 3 client:

EXECUTE PROCEDURE *bignum* (65535, 65535) returns *ERROR* can't access INT64 when executed by a dialect 1 client.

EXECUTE PROCEDURE *bignum* (65535, 65535) returns 4294836225 when executed by a dialect 3 client.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Generators

Firebird 1 generators return a 64-bit value, and wrap around only after $2^{32} - 1$ positive invocations (assuming an increment of 1), rather than after $2^{16} - 1$ as in older databases. Applications should use an *ISC_INT64* variable to hold the value returned by a generator if it is likely to extend beyond the long integer range in the lifetime of the application. A client using dialect 1 receives only the least significant 32 bits of the updated generator value, but the entire 64-bit value is incremented by the engine even when returning a 32-bit value to a client that uses dialect 1. If your database was using an INTEGER field for holding generator values, you may need to recreate the field so that it can hold 64-bit integer values.

To populate a field that uses a generator, in Firebird 1 should you define the column as INTEGER, or as NUMERIC(18) if you need to cater for values beyond the long integer range.

- ★ You may even accommodate a generator value in a SMALLINT column if you are certain it will never extend past 255.

Miscellaneous issues

- IN clauses have a limit of 1,500 elements

Resolution If you have more than 1,500 elements, accommodate the values in a table designed for volatile use and handle the criteria with a different style of predicate.

- Arithmetic operations on character fields are no longer permitted in client dialect 3

Resolution Explicitly cast the information before performing arithmetic calculations.

- Using `isql` to select from a TIMESTAMP column displays all information when client dialect is 3.

Resolution In older versions, the time portion of a timestamp displayed only if SET TIME ON was in effect. In Firebird 1 client dialect 3, the time portion of the timestamp always displays.

Migrating servers and databases

You can migrate your servers and applications to Firebird 1 at different times. They are separate migrations.

Bear the following issues in mind as you plan your migration:

- When you migrate a server to Firebird 1, you must also migrate all the databases on that server.
- Older clients can still access databases that have been migrated to Firebird 1. You must be aware, however, that they cannot access new datatypes or data stored as INT64, and they always handle double quoted material as strings.

**Symbols** [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

- InterBase strongly recommends that you establish a migration testbed to check your migration procedures before migrating production servers and databases. The test bed does not need to be on the same platform as the production clients and servers that you are migrating.

The migration path varies somewhat depending on whether you are replacing an existing server or installing a new server and moving old databases there. Upgrading an existing server costs less in money, but may cost more in time and effort. The server and all the databases that must migrate with it are unavailable during the upgrade. If you have hardware available for a new Firebird 1 server, the migration can be done in parallel, without interrupting service more than very briefly. This option also offers an easier return path if problems arise with the migration.

“In-place” server migration

This section describes the recommended steps for replacing an InterBase 5 server with a Firebird 1 server.

- 1 Shut down each database before backup to ensure that no transactions are in progress.
- 2 Back up all databases on the InterBase 5 server. Include **isc4.gdb** if you want to preserve your configured user IDs.

As a precaution, you should validate your databases before backing up and then restore each database to ensure that the backup file is valid.

- 3 Shut down the InterBase 5 server. If your current server is InterBase SuperServer, you are not required to uninstall the server if you intend to install over it, although uninstalling is always good practice. You cannot have multiple versions of InterBase on the same machine. If your current server is InterBase Classic, you *must* uninstall before installing Firebird 1.
- 4 Install the Firebird 1 server.

Note The install does not overwrite **isc4.gdb** or **isc4.gbk**.

- 5 Start the new server.
 - On Windows NT or Windows 2000, go to Services in the Control Panel and start the InterBase Manager.
 - On Windows 9x, go to the Control Panel and run the InterBase Manager applet.
 - On UNIX platforms, issue the following command to start the server as user “interbase”:

```
# echo "/usr/interbase/bin/ibmgr -start -forever" | su interbase
```



Symbols **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

Note InterBase can run only as user "root" or user "interbase" on UNIX.

6 To restore the list of valid users, follow these steps:

- a** Restore **isc4.gbk** to **isc4_old.gdb**
- b** Shut down the server
- c** Copy **isc4_old.gdb** over **isc4.gdb**
- d** Copy **isc4_old.gbk** over **isc4.gbk**
- e** Restart the server

7 Delete each ODS 9 database file. Restore each database from its backup file. This process creates Firebird 1, ODS 10, dialect 1 databases.

8 Perform a full validation of each database.

Migrating to a new server

This section describes the recommended steps for installing Firebird 1 as a new server and then migrating databases from a previous InterBase 5 installation. The process differs only slightly from an in-line install.

1 Back up all databases on the InterBase 5 server. Include **isc4.gdb** if you want to preserve your configured user IDs. Shut down the databases before backup to ensure that no transactions are in progress.

2 Install the Firebird 1 server.

3 Start the new Firebird 1 server.

- On WindowsNT, go to Services in the Control Panel and start InterBase Manager.
- On Windows 9x, got to the Control Panel and run the InterBase Manager applet.
- On UNIX platforms, issue the following command to start the server as user "interbase":

```
# echo "/usr/interbase/bin/ibmgr -start -forever" | su interbase
```

 InterBase can run only as user "root", user firebird or user "interbase" on Linux/UNIX.

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

- 4 Copy the database backup files to the new server and restore each database from its backup file. This process creates Firebird 1, ODS 10, dialect 1 databases.

Save your backup files until your migration to dialect 3 is complete.
- 5 To restore the list of valid users, follow these steps:
 - a Restore `isc4.gbk` to `isc4_old.gdb`
 - b Shut down the server
 - c Copy `isc4_old.gdb` over `isc4.gdb`
 - d Copy `isc4_old.gbk` over `isc4.gbk`
 - e Restart the server
- 6 Perform a full validation of each database on the new server.

After performing these steps, you have an Firebird 1 server and Firebird 1, dialect 1 databases.

About Firebird 1, dialect 1 databases

When you back up an InterBase 5 database and restore it in Firebird 1, what do you have?

- An InterBase 5 client can access everything in the database with no further changes.
- If there are object names—column or table names, for instance—that include any of the new reserved words, you must change these names in order to access these objects with a Firebird 1 dialect 1 client. The new `ALTER COLUMN` clause of `ALTER TABLE` makes it easy to implement column name changes.
- InterBase 5 clients can still access the columns.
- Dialect 3 clients can access these columns as long as they delimit them with double quotes.
- The words listed in [New reserved words](#) are reserved words. However, the new datatypes `TIME` and `DATE` are not available to use as datatypes. `DATE` columns have the old meaning—both date and time. The new meaning of `DATE`—date only—is available only in dialect 3.
- All columns that were previously `DATE` datatype are now `TIMESTAMP` datatype. `TIMESTAMP` contains exactly the information that `DATE` did in previous versions.

**Symbols** [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

- Exact numeric columns—those that have a DECIMAL or NUMERIC datatype with precision greater than 9—are still stored as DOUBLE PRECISION datatypes. All arithmetic algorithms that worked before on these columns still work as before. It is not possible to store data as INT64 in dialect 1.

Transforming databases to Dialect 3

There are four major areas of concern when migrating a database from dialect 1 to dialect 3: double quotes, the DATE datatype, large exact numerics (for purposes of this discussion, NUMERIC and DECIMAL datatypes that have a precision greater than 9), and reserved words.

The process varies somewhat depending on whether you can create an application to move data from your original database to an empty dialect 3 database. If you do not have access to such a utility, you need to perform an in-place migration of the original database.

Overview

In either method, you begin by extracting the metadata from your database, examining it for problem areas, and fixing the problems.

- If you are performing an in-place migration, you copy corrected SQL statements from the metadata file into a new script file, modify them, and run the script against the original database. Then you set the database to dialect 3. There are some final steps to take in the dialect 3 database to store old data as INT64.
- If you have a utility for moving data from the old database to a newly created empty database, you use the modified metadata file to create a new dialect 3 database and use the utility to transfer data from the old database to the new.

In both cases, you must make changes to the new database to accommodate migrated columns that must be stored as INT64 and column constraints and defaults that originally contained double quotes.

The two methods are described below.

Method one: in-place migration

- If you have not migrated the database to Firebird 1, dialect 1, do so first. Back up the database again.
- Extract the metadata from the database using `isql -x`. If you are migrating legacy databases that contain GDML, see [Migrating very old databases](#).

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

- 3 Prepare an empty text file to use as a script file. As you fix data structures in the metadata files, you will copy them to this file to create a new script.

Note You could also proceed by removing unchanged SQL statements from the original metadata file, but this is more likely to result in problems from statements that were left in error. InterBase recommends creating a new script file that contains only the statements that need to be run against the original database.

For the remaining steps, use a text editor to examine and modify the metadata and script files. Place copied statements into the new script file in the same order they occur in the metadata file to avoid dependency errors.

- 4 Search for each instance of double quotes in the extracted metadata file. These can occur in triggers, stored procedures, views, domains, table column defaults, and constraints. Change each double quote that delimits a string to a single quote. Make a note of any tables that have column-level constraints or column defaults in double quotes.

Copy each changed statement to your empty script file, but do not copy ALTER TABLE statements whose only double quotes are in column-level constraints or column defaults.

Important When copying trigger or stored procedure code, be sure to include any associated SET TERM statements.

Quoted quotes If there is any chance that you have single or double quotes *inside* of strings, you must search and replace on a case-by-case basis to avoid inappropriate changes. The handling of quotation marks within strings is as follows:

TABLE 27–2 Handling quotation marks inside of strings

<i>String:</i>	In "peg" mode
<i>Double-quoted:</i>	not allowed
<i>Single-quoted:</i>	' In "peg" mode '
<i>String:</i>	O'Reilly



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

TABLE 27–2 Handling quotation marks inside of strings

Double-quoted: not allowed

Single-quoted: 'O' 'Reilly'

- 5 In the new script file, search for occurrences of the TIMESTAMP datatype. In most cases, these were DATE datatypes in your pre-6 database. For each one, decide whether you want it to be TIME, TIMESTAMP, or DATE in your dialect 3 database. Change it as needed.
- 6 Repeat step 5 in the metadata file. Copy each changed statement to your new script file.
- 7 In the new script file, search for occurrences of reserved words that are used as object names and enclose them in double quotes; that makes them delimited identifiers.
- 8 Repeat step 7 in the metadata file. Copy each changed statement to your new script file.
- 9 In each of the two files, search for each instance of a DECIMAL or NUMERIC datatype with a precision greater than 9. Consider whether or not you want data stored in that column or with that domain to be stored as DOUBLE PRECISION or INT64. See [Do you really need to migrate your NUMERIC and DECIMAL datatypes?](#) for a discussion of issues. For occurrences that should be stored as DOUBLE PRECISION, change the datatype to that. Leave occurrences that you want stored as INT64 alone for now. Copy each changed statement that occurs in the metadata file to your new script file.

Perform the following steps in your new script file:

- 10 Locate each CREATE TRIGGER and CREATE DOMAIN statement and change it to ALTER TRIGGER or ALTER DOMAIN as appropriate.
- 11 Locate each CREATE VIEW statement. Precede it by a corresponding DROP statement. For example, if you have a CREATE VIEW *foo* statement, put a DROP VIEW *foo* statement right before it, so that when you run this script against your database, each view first gets dropped and then re-created.
- 12 If you have any ALTER TABLE statements that you copied because they contain named table-level constraints, modify the statement so that it does nothing except drop the named constraint and then add the constraint back with the single quotes.

**Symbols** [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

- 13 Check that stored procedure statements are ALTER PROCEDURE statements. This should already be the case.
- 14 At the beginning of the script, put a CONNECT statement that connects to the original database that you are migrating.
- 15 Make sure your database is backed up and run your script against the database.
- 16 Use **gfix** to change the database dialect to 3.

```
gfix -sql_dialect 3 database.gdb
```

Note To run **gfix** against a database, you must attach as either the database owner or SYSDBA.

- 17 At this point, DECIMAL and NUMERIC columns with a precision greater than 9 are still stored as DOUBLE PRECISION. To store the data as INT64, follow the steps in [DECIMAL and NUMERIC datatypes](#).

- 18 Use **gfix** to validate the database.

That's it. You've got a dialect 3 database. There is a little more work to do if you want your NUMERIC and DECIMAL columns with a precision of greater than 9 to be stored as INT64. At this point, they are still stored as DOUBLE PRECISION. To decide whether you want to change the way data in these columns is stored, read [DECIMAL and NUMERIC datatypes](#).

In addition, there are some optional steps you can take that are described in the following sections, [Column defaults and column constraints](#) and [Unnamed table constraints](#).

Important If you ever extract metadata from the dialect 3 database that you created using the steps above, and if you plan to use that metadata to create a new database, check to see if the extracted metadata contains double quotes delimiting string constants in column defaults, column constraints, or unnamed table constraints. Change any such occurrences to single quotes before using the metadata to create the new database.

Column defaults and column constraints

The steps above permitted you to retain double quoted string constants in column defaults, column constraints, and unnamed table constraints. This is possible because, once created, InterBase stores them in binary form.

Following the steps above creates a dialect 3 database that is fully functional, but if it contains double quoted string constants in column defaults, column constraints, or unnamed column constraints, inconsistencies are

**Symbols** [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

visible when you SHOW metadata or extract it. You can choose to resolve these inconsistencies by following these steps:

- 1 Back up the database.
- 2 Examine the metadata to detect each occurrence of a column default or column constraint that uses double quotes.
- 3 For each affected column, use the ALTER COLUMN clause of the ALTER TABLE statement to give the column a temporary name. If column position is likely to be an issue with any of your clients, change the position as well.
- 4 Create a new column with the desired datatype, giving it the original column name and position.
- 5 Use UPDATE to copy the data from old column to the new column:

```
UPDATE table_name
      SET new_col = old_col;
```

- 6 Drop the old column.

Unnamed table constraints

Read the first two paragraphs under [Column defaults and column constraints](#) to understand why you don't always need to change constraints with double quotes to single-quoted form, and why you might want to change them.

To bring unnamed table constraints that contain double quotes into compliance with the dialect 3 standard, follow these steps:

- 1 Back up the database.
- 2 Examine the metadata to detect each occurrence of an unnamed table constraint that uses double quotes.
- 3 For each occurrence, use SHOW TABLE to see the name that InterBase has assigned to the constraint.
- 4 Use ALTER TABLE to drop the old constraint, using the name given in the SHOW TABLE output and add a new constraint. For ease in future handling, give the constraint a name.

**Symbols** **Numerics** **A** **B** **C** **D** **E** **F** **G** **H** **I** **J** **K** **L** **M** **N** **O** **P** **Q** **R** **S** **T** **U** **V** **W** **X** **Y** **Z**

If SHOW TABLE shows that InterBase stores the unnamed constraint as "INTEG_2", then issue the following statement to change the constraint:

```
ALTER TABLE foo
    DROP CONSTRAINT INTEG_2,
    ADD CONSTRAINT new_name
        CHECK (col_name IN ('val1', 'val2', 'val3'));
```

Converting NUMERIC and DECIMAL datatypes

If you back up a NUMERIC or DECIMAL column with a precision greater than 9 (for example, NUMERIC(12,2)) in an InterBase 5 or earlier database and restore the database as Firebird 1, the column is still stored as DOUBLE PRECISION. Because InterBase does not allow datatype conversions that could potentially result in data loss, you cannot use the ALTER COLUMN statement to change the column datatype from DOUBLE PRECISION to INT64. To migrate a DOUBLE PRECISION column to an INT64 column, you must create a new INT64 column and copy the contents of the older column into it.

In Firebird 1 dialect 3, when you create a NUMERIC or DECIMAL column with a precision greater than 9, data in it is automatically stored as an INT64 exact numeric.

If you want NUMERIC and DECIMAL datatypes with a precision greater than 9 to be stored as exact numerics, you must take some extra steps after migrating to dialect 3. The following sections tell you how to decide whether you really need to take these steps and how to perform them if you decide you want the exact numerics.

DO YOU REALLY NEED TO MIGRATE YOUR NUMERIC AND DECIMAL DATATYPES?

Future versions of InterBase will no longer support dialect 1. It is offered now as a transitional mode. As you migrate your databases to dialect 3, consider the following questions about columns defined with NUMERIC and DECIMAL datatypes:

- Is the precision less than 10? If so, there is no issue. You can migrate without taking any action and there will be no change in the database and no effect on clients.
- For NUMERIC and DECIMAL columns with precision greater than 9, is DOUBLE PRECISION an appropriate way to store your data?

**Symbols** [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

- In many cases, the answer is “yes.” If you want to continue to store your data as DOUBLE PRECISION, change the datatype of the column to DOUBLE PRECISION either before or after migrating your database to dialect 3. This doesn’t change any functionality in dialect 3, but it brings the declaration into line with the storage mode. In a dialect 3 database, newly-created columns of this type are stored as INT64, but migrated columns are still stored as DOUBLE PRECISION. Changing the declaration avoids confusion.
- DOUBLE PRECISION may not be appropriate or desirable for financial applications and others that are sensitive to rounding errors. In this case, you need to take steps to migrate your column so that it is stored as INT64 in dialect 3. As you make this decision, remember that INT64 does not store the same range as DOUBLE PRECISION. Check whether you will experience data loss and whether this is acceptable.

MIGRATING NUMERIC AND DECIMAL DATATYPES

Read [Do you really need to migrate your NUMERIC and DECIMAL datatypes?](#) to decide whether you have columns in a dialect 1 database that would be best stored as 64-bit integers in a dialect 3 database. If this is the case, follow these steps for each column:

- 1 Migrate your database to Firebird 1 as described in [Method one: in-place migration](#).
- 2 Use the ALTER COLUMN clause of the ALTER DATABASE statement to change the name of each affected column to something different from its original name. If column position is going to be an issue with any of your clients, use ALTER COLUMN to change the positions as well.
- 3 Create a new column for each one that you are migrating. Use the original column names and if necessary, positions. Declare each one as a DECIMAL or NUMERIC with precision greater than 9.
- 4 Use UPDATE to copy the data from each old column to its corresponding new column:

```
UPDATE tablename  
      SET new_col = old_col;
```

- 5 Check that your data has been successfully copied to the new columns and drop the old columns.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Method two: migrating to a new database

If you can create a data transfer utility that copies data between databases, the process of migrating a database to dialect 3 is considerably simplified.

Overview Extract the metadata from your database, examine it for problem areas, and fix the problems. Use the modified metadata file to create a new dialect 3 database and use an application to transfer data from the old database to the new.

- 1 If you have not migrated the database to Firebird 1, dialect 1, do so first. Back up the database again.
- 2 Extract the metadata from the database using **isql -x**. If you are migrating a database that contains data structures created with GDML, see [Migrating very old databases](#).

For the following steps, use a text editor to examine and modify the metadata file.

- 3 Search for each occurrence of the TIMESTAMP datatype. In most cases, these were DATE datatypes in your pre-6 database. Decide whether you want it to be TIME, TIMESTAMP, or DATE in your dialect 3 database. Change it as needed.
- 4 Find all instances of reserved words that are used as object names and enclose them in double quotes to make them delimited identifiers.
- 5 Search for each instance of double quotes in the extracted metadata file. These can occur in triggers, stored procedures, views, domains, exceptions, table column defaults, and constraints. Change each double quote to a single quote.
- 6 Search for each instance of a DECIMAL or NUMERIC datatype with a precision greater than 9. Consider whether or not you want that data stored as DOUBLE PRECISION or INT64. See [Do you really need to migrate your NUMERIC and DECIMAL datatypes?](#) for a discussion of issues. For occurrences that should be stored as DOUBLE PRECISION, change the datatype to that. Leave occurrences that you want stored as INT64 alone for now.
- 7 At the beginning of the file, enter SET SQL DIALECT 3. On the next line, uncomment the CREATE DATABASE statement and edit it as necessary to create a new database.
- 8 Run the metadata file as a script to create a new database.
- 9 Use your data transfer utility to copy data from the old database to the new dialect 3 database. In the case of a large database, allow significant time for this.



Symbols [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

10 Validate the database using `gfix`.

11 At this point, DECIMAL and NUMERIC columns with a precision greater than 9 are still stored as DOUBLE PRECISION. To store the data as INT64, follow the steps in [Converting NUMERIC and DECIMAL datatypes](#).

Migrating very old databases

If you have legacy databases in which some data structures were created with GDML, you may need to extract metadata in a slightly different way.

- 1 Try extracting metadata as described in Step 2 above and examine it to see if all tables and other DDL structures are present. If they are not, delete the metadata file and extract using the `-a` switch instead of the `-x` switch. This extracts objects created in GDML.
- 2 You may have to change some of the code to SQL form. For example, the following domain definition

```
CREATE DOMAIN NO_INIT_FLAG AS SMALLINT  
  ( no_init_flag = 1 or  
    no_init_flag = 0 or  
    no_init_flag missing );
```

needs to be translated to:

```
CREATE DOMAIN NO_INIT_FLAG AS SMALLINT  
  CHECK ( VALUE = 1 OR VALUE = 0 OR VALUE IS NULL );
```

- 3 Some code may be commented out. For example:

```
CREATE TABLE BOILER_PLATE (BOILER_PLATE_NAME NAME,  
  DATE DATE,  
  CREATED_DATE COMPUTED BY /* Date */);
```

needs to be changed to:

```
CREATE TABLE BOILER_PLATE (BOILER_PLATE_NAME NAME,  
  "DATE" DATE,  
  CREATED_DATE COMPUTED BY "DATE");
```



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Migrating clients

To migrate an older client application to Firebird 1, install the Firebird 1 client onto the platform where the client application resides. An InterBase server then recognizes that client as a Firebird 1 dialect 1 client. It is good practice to recompile and relink the application and make note of field names, datatype use, and so on in the new application. When you recompile, state the dialect explicitly:

```
SET SQL DIALECT n;
```

- ❖ If you have databases that use any of the new Firebird 1 reserved words as object identifiers and you are not migrating those databases to dialect 3, you might want to not migrate your InterBase 5 clients. If you migrate them to Firebird 1 dialect 1, you lose the ability to access those reserved word columns. See [New reserved words](#).

When you recompile an existing **gpre** client, you must recompile it with the **gpre -sql_dialect n** switch.

- To write embedded SQL applications that address all Firebird dialect 3 functionality, compile them using **gpre -sql_dialect 3**.

TABLE 27-3 Migrating clients: summary

Client	How to migrate
Older applications such InterBase® 5 applications	<ul style="list-style-type: none">• Dialect is 1; there is no way to change the dialect• An InterBase 5 client application becomes a Firebird dialect 1 client whenever the Firebird client is installed on the machine with the client
ISQL	<ul style="list-style-type: none">• Issue the command line option: <code>-sql_dialect n</code>• Or issue the command <code>SET SQL DIALECT n;</code>



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

TABLE 27-3 Migrating clients: summary

Client	How to migrate
GPRE	<ul style="list-style-type: none">• Issue the command line option <code>-sql_dialect n</code>• Or issue the command <pre>EXEC SQL SET SQL DIALECT n;</pre>
BDE	BDE version 5.2 supports Firebird, but has some outstanding bugs with date/time and some numerical types.. To access InterBase dialect 3 features from Delphi and C++Builder, consider using a direct-to-API component set such as IB Objects, FIBPlus or IBX.
InterClient	Applications can use dialect 3 if upgraded to use the new Firebird Java drivers.
Direct API calls	Set the dialect parameter on <code>isc_dsql_execute_immediate()</code> , <code>isc_dsql_exec_immed2()</code> , <code>isc_dsql_prepare()</code> API calls to the desired dialect value: 1 or 3

Migrating data from unrelated DBMS products

If you have a large amount of data in another DBMS such as Paradox, the most efficient way to bring the data into InterBase is to export the data from the original DBMS into InterBase external file format. see [Uses for the EXTERNAL FILE option](#) on p. 336 of chapter 17 for more information about InterBase external files. Then insert the data from the external files into the internal tables. It is best not to have any constraints on new internal tables; you can validate the database more easily once the data is in InterBase. If constraints do exist, you will need triggers to massage the incoming data.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

CHAPTER 28

Glossary

Term	Definition
Aggregate (function)	An aggregate function returns a result derived from a calculation that aggregates (collects together) values from a set of rows that is grouped in some fashion by the syntax of the SQL statement. For example, the internal function SUM() operates on a non-null numerical column to return the result of adding all of the values in that column together for all of the rows cited by a WHERE or GROUP BY clause. Output which is aggregated by a WHERE clause returns one row of output, whereas that aggregated by a GROUP BY clause potentially returns multiple rows.
Alerter (events)	<i>Alerter</i> is a term coined to represent a client routine or class which is capable of “listening” for specific database EVENTS generated from triggers or stored procedures running on the server.
'ALICE'	Internal name for the source code for the <code>gfix</code> utilities - a corruption of the words “all else”.
Alternative key ('alternate key')	This is a term used for a unique key that is not the primary key. A unique key is created by applying the UNIQUE constraint to a column or group of columns. A foreign key in a formal referential integrity relationship can link its REFERENCES clause to an alternative key.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Term	Definition
Application Programming Interface (API)	An application programming interface (API) provides a set of formal structures through which application programs can communicate with functions internal to another software. The Firebird API surfaces such an interface as a client library compiled specifically for each supported platform. The Firebird API structures are C structures, designed to be translatable to virtually any host application language. Translations can be found for Java, Pascal, Perl and, to some degree, PHP 4, Python and others.
Argument	An <i>argument</i> is a value of a prescribed data type and size which is passed to a function or stored procedure to be operated upon. Stored procedures can be designed to both accept <i>input</i> arguments and return <i>output</i> arguments. For the returned values of functions (both internal and user-defined) the term <i>result</i> is more commonly used than <i>argument</i> . The terms <i>parameter</i> and <i>argument</i> are often used interchangeably with regard to stored procedures, thanks to Borland's adoption of the term <i>parameter</i> in its Delphi data access classes to name the properties to which stored procedure arguments are assigned.
Array slice	A contiguous range of elements from a Firebird array is known as a <i>slice</i> . A slice can consist of any contiguous block of data from an array, from a single element of one dimension to the maximum number of elements of all defined dimensions.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Term	Definition
Atomicity	In the context of transactions, atomicity refers to the character of the transaction mechanism that wraps a grouping of changes to rows in one or more tables to form a single unit of work that is either committed entirely or rolled back entirely. In the context of a key, a key is said to be atomic if its value has no meaning as data.
AutoCommit	When a change is posted to the database, it will not become permanent until the transaction in which it was posted is committed by the client application. If the client rolls back the transaction, instead of committing it, the posted changes will be cancelled. Some client development tools provide a mechanism by which posting any change to any table invokes a follow-up call to commit the transaction, without further action by the user. This mechanism is usually called <i>AutoCommit</i> , or some similar term. It is not a Firebird mechanism—Firebird never commits transactions started by clients.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Term	Definition
Backup/restore (Firebird-style)	<p><i>Backup</i> is an external process initiated by a user—usually SYSDBA—to decompose a database into a collection of compressed disk structures comprising metadata and data which are separated for storage. <i>Restore</i> is another external process—also user-initiated—which completely reconstructs the original database from these stored elements.</p> <p>The backup process also performs a number of housecleaning tasks on a database whilst reading it for backing up; and a restored database will be completely free of “garbage”.</p> <p>See also <code>gbak</code>.</p>
BDE	<p>Borland Database Engine. Originally designed as the database engine of Paradox, it was extended to provide a generic middleware connectivity layer between a variety of relational database engines and Borland application development tools for the Microsoft DOS and Windows platforms. The vendor-specific rules applicable to each RDBMS are encapsulated in a set of driver libraries known as SQL Links. The SQL Links drivers are version-specific.</p> <p>By the time Borland released InterBase 6, the codebase from which Firebird was developed, it had already deprecated the BDE in favour of more modern driver technologies.</p>



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Term	Definition
BDE (cont.)	The last known version of the BDE (v.5.2) shipped with Borland Delphi 6 to provide legacy support, especially for ISAM databases. An InterBase 6 driver in this distribution only partly supports Firebird and InterBase 6 dialect 3. Developers building Firebird applications in Borland Delphi and C++Builder generally prefer to avoid the BDE and use direct-to-API replacement components such as IB Objects, FreeIBPlus, SQL-Direct or Borland InterBaseXpress.
Binary tree	A logical tree structure in which a node can subtend a maximum of two branches. Firebird indexes are formed in binary tree structures.
BLOB	Mnemonic for Binary Large Object. This is a data item of unlimited size, which may be in any format, which is streamed into the database byte-by-byte and stored without any format modification. Firebird can distinguish BLOBS of different types by means of <i>sub-types</i> . Firebird's ancestor, InterBase, was the first relational database to support storage of BLOBS. See also <i>CLOB</i> .
BLOB Control Structure	A C structure, declared in a UDF module as a <code>typedef</code> , through which a blob UDF accesses a blob. A blob UDF cannot refer to actual BLOB data: a pointer to a blob control structure instead.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Term	Definition
BLOB Filter	A BLOB Filter is a specialized UDF that transforms BLOB data from one subtype to another. Firebird includes a set of internal BLOB filters which it uses in the process of storing and retrieving metadata. One internal filter converts text data transparently between SUB_TYPE 0 (none) and SUB_TYPE 1 (text, sometimes referred to as 'Memo').
BLR	Binary Language Representation, an internal relational language with a binary notation that is a super set of the "human-readable" languages that can be used with Firebird, viz.. SQL and GDML. Firebird's DSQL interface to the server translates queries into BLR. The BLR of compiled triggers and stored procedures, check constraints, defaults and views are stored in BLOBS. Some client tools, for example IB_SQL and the command-line tool isql , have facilities to inspect this BLR code. isql : execute the command SET BLOB ALL, perform SELECT statements to get the appropriate BLR fields from the system tables.
Buffer	A buffer is a block of memory for caching copies of pages read from the database. The term <i>buffer</i> is synonymous with <i>cache page</i> .
'BURP'	Internal name for the gbak code—a mnemonic for “Backup [and] Restore Program”.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Term	Definition
Cache	<p>When a page is read from disk it is copied into a block of memory known as the <i>database cache</i> or, simply, the <i>cache</i>. The cache consists of <i>buffers</i>, each the size of a database page, determined by the <i>page_size</i> parameter declared when the database is created.</p> <p>The word <i>buffer</i> in this context means a block of memory exactly the same size as one database page. The cache size is configurable, as a number of pages (or buffers). Hence, to calculate the size of the cache, multiply the <i>page_size</i> by the number of cache pages (buffers).</p>
Cardinality (of an output set)	<p>The number of rows output by a SELECT statement. The <i>cardinality of a row</i> refers to its position in the top-to-bottom order of the output set.</p>
Case-insensitive index	<p>This is an index using a collation order in which lower-case characters are treated as though they were the same as their upper-case equivalents. Firebird 1.0 does not support case-insensitive indexes.</p>
Cascading integrity constraints	<p>Firebird provides the optional capability to prescribe specific behaviors and restrictions in response to requests to update or delete rows in tables which are pointed to by the REFERENCES sub-clause of a FOREIGN KEY constraint. The CASCADE keyword causes changes performed on the “parent” row to flow on to rows in tables having the FOREIGN KEY dependencies. ON DELETE CASCADE, for example, will cause dependent rows to be deleted when the parent is deleted.</p>



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Term	Definition
Casting	Casting is a mechanism for converting output or variable values of one data type into another data type by means of an expression. Firebird SQL provides the CAST() function for use in both dynamic SQL (DSQL) and procedural SQL (PSQL) expressions.
Character set	Two super-sets of printable character images and control sequences are in general use today in software environments—ASCII and UNICODE. ASCII characters, represented by one byte, have 256 possible variants whilst UNICODE, of two, three or four bytes, can accommodate 65536 possibilities. Because databases need to avoid the prohibitive overhead of making available every possible printable and control character used for programming anywhere in the world, the super-sets are divided up into <i>code pages</i> , also known as <i>code tables</i> . Each code page defines a subset of required characters for a specific language, or family of languages, mapping each character image to a number. The images and control sequences within each code page are referred to collectively as a character set. A character image might be mapped to different numbers in different characters sets. Firebird supports a default character set for a database and definition of an explicit character set for any character, varchar or BLOB SUB_TYPE 1 (text BLOB) column. If no character set is defined for a database, its character set defaults to NONE, causing all character data to be stored exactly as presented, with no attempt to convert characters (transliterate) to any particular character set.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Term	Definition
Classic architecture	The term "Classic architecture" refers to the older InterBase versions predating the introduction of the "Superserver" model in InterBase 5 for Windows and in Firebird for some Unix-style platforms, including Linux. Classic was revived as an alternative for several of the Unix-style platforms.
CLOB	Mnemonic for 'Character Large OBject'. This term has crept into use recently, as other RDBMSs have refined their support for storing large objects in databases. A CLOB is equivalent to Firebird's BLOB SUB_TYPE 1, i.e. it is a BLOB structure which has special handling for storing and reading streams of bytes which are plain text characters. See also <i>BLOB</i> .
Coercing data types	In the Firebird API's XSQLDA structures, converting a data item of one SQL type to another, compatible SQL type is known as <i>data type coercion</i> .
Collation order	Defines how a sort operation orders character columns in output sets, the pairing of lower case and upper case characters for the UPPER() function and how characters in character columns are compared in the WHERE and HAVING clauses of a SELECT statement. A collation order applies to a specific character set. If multiple collation orders are available for a particular character set, one collation order will be treated as the default.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Term	Definition
Column	In SQL databases, data are stored in structures which can be retrieved as tables or, more correctly, <i>sets</i> . A set consists of one or more rows each identical in the horizontal arrangement of distinctly defined items of data. One distinct item of data considered vertically for the length of the set is known as a <i>column</i> . Application developers often refer to columns as <i>fields</i> .
Commit	When applications post changes affecting rows in database tables, new versions of those rows are created in temporary storage blocks. Although the work is visible to the transaction in which it occurred, it cannot be seen by other users of the database. The client program must instruct the server to commit the work in order for it to be made permanent. If a transaction is not committed, it must be rolled back in order to undo the work.
Concurrency	The term <i>concurrency</i> broadly refers to multiple users accessing the same data simultaneously. It is also widely used in documentation and support lists to refer to the particular set of attributes that apply to a transaction—isolation level, locking policy and others. For example, someone may ask you “What are your concurrency settings?” Even more specifically, the word <i>concurrency</i> is sometimes used as a synonym for the SNAPSHOT isolation level.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Term	Definition
Constraint	Firebird makes many provisions for defining formal rules which are to be applied to data. Such formal rules are known as <i>constraints</i> . For example, a PRIMARY KEY is a constraint which marks a column or group of columns as a database-wide pointer to all of the other columns in the row defined by it. A CHECK constraint sets one or more rules to limit the values which a column can accept.
Contention	When two transactions attempt to update the same row of a table simultaneously, they are said to be <i>in contention</i> —they are contending (or competing).
Correlated Subquery	A query specification can define output columns that are derived from expressions. A sub-query is a special kind of expression that returns a single value which is itself the output of a SELECT statement. In a correlated sub-query, the WHERE clause contains a search criterion which is derived from the value of a column of the table from which the outer SELECT is specified.
Crash	A slang term for abnormal termination of the server or a client application.
Crash recovery	Processes or procedures that are implemented to restore the server and/or client applications into running condition following an abnormal termination of either the server or the application, or both.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Term	Definition
Cyclic links	In a database context, this is a term for an inter-table dependency where a foreign key in one table(TableA) refers to a unique key in another table (TableB) which contains a foreign key which points back, either directly, or through a reference to another table, to a unique key in TableA.
Database	In its broadest sense, the term <i>database</i> applies to any persistent file structure which stores data in some format which permits it to be retrieved and manipulated by applications.
DDL	Mnemonic for <i>Data Definition Language</i> —the subset of SQL that is used for defining and managing the structure of data objects.
Deadlock	When two transactions are in contention to update the same version of a row and both transactions are set to NO WAIT, neither transaction has precedence over the other and the transactions are said to be <i>in deadlock</i> . Client application code must resolve a deadlock by having one transaction roll back in order to allow the other to commit its work.
Degree (of an output set)	The number of columns output by a SELECT statement. The <i>degree of a column</i> refers to its position in the left-to-right sequence of columns in a set.
Deployment	The process of distributing the components of your application system to its intended users for production use.
DML	Mnemonic for <i>Data Manipulation language</i> , the major subset of SQL statements, which perform operations on sets of data.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Term	Definition
Domain	A Firebird SQL feature whereby you can assign an identifying name to a set of data characteristics and constraints (CREATE DOMAIN) and then use this name in lieu of a data type when defining columns in tables.
DPB	Mnemonic for <i>Database Parameter Buffer</i> , a character array used by applications to communicate the parameters defining the characteristics of a client connection request, along with their specific item values, across Firebird's application programming interface (API).
DSQL	Dynamic SQL—refers to statements that an application submits in run-time, with or without parameters, as contrasted with "static SQL" statements which are coded directly into special code blocks of a host language program and are subsequently preprocessed for compilation within embedded SQL applications. Applications which use Firebird API calls, either directly or through a class library which encapsulates the Firebird API, are using DSQL.
'DUDLEY'	Internal name for the source-code for the deprecated metadata utility, <code>gdef</code> . The name derives from the mnemonic "DDL" (Database Definition Language).
dyn, also DYN	A byte-encoded language for describing data definition statements. Firebird's DSQL subsystem parses data definition language (DDL) statements and passes them to a component that outputs DYN for interpretation by the Y-Valve, another subsystem which is responsible for updating the system tables.



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

Term	Definition
Error	A condition in which a requested SQL operation cannot be performed because something is wrong with the data supplied to a statement or procedure or something is wrong with the syntax of the statement itself. When Firebird encounters an error, it does not proceed with the request but returns an exception message to the client application. See also Exception.
Error Code	An integer constant returned to the client or to a calling procedure when Firebird encounters an error condition. See also Error, Exception.
ESQL	Mnemonic for Embedded SQL, the subset of SQL provided for embedding static SQL in special blocks within a host application.
Event	An event realizes Firebird's capability to pass a notification to a "listening" client application if requested to do so by a POST_EVENT call in a trigger or stored procedure.
Exception	An exception is the Firebird server's response to an error condition occurring during a database operation. Several hundred exception conditions are realized as error codes, in a variety of categories, which are passed back to the client in the Error Status Vector (Array). Exceptions are also available to stored procedures and triggers, where they can be handled by a custom routine. Firebird supports user-defined exceptions as well.

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

Term	Definition
Execute	<p>In a client application, <i>execute</i> is commonly used as a verb which means “perform my request” when a data manipulation statement or stored procedure call has been prepared by the client application.</p> <p>In DSQL, the phrase EXECUTE PROCEDURE is used with a stored procedure identifier and its input arguments, to invoke executable stored procedures.</p>
FIBPlus	Mnemonic for the extended, commercial version of the older FreeIBComponents of Greg Deatz, data access components encapsulating the Firebird and InterBase APIs, for use with Borland’s Client/Server or Enterprise editions of Delphi and C++ Builder.
Foreign Key	<p>A foreign key is a column or group of columns in one table that points to a corresponding primary or unique key in another table. If the foreign key is non-unique and the table itself has a declared primary key, the table is capable of being the “many” side of a 1:Many relationship.</p> <p>Firebird supports the declaration of a formal <i>foreign key constraint</i> which will enforce referential integrity automatically. When such a constraint is declared, Firebird automatically creates a non-unique index on the column or columns to which the constraint applies and also records the dependency between the table and the table whose primary or unique key is referred to by the REFERENCES clause of the FOREIGN KEY declaration.</p>



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Term

Definition

Garbage collection

General term for the cleanup process that goes in the database during normal use to remove obsolete back-versions of rows that have been updated. In Superserver, garbage collection runs as a background thread to the main server process.

Garbage collection can also be performed by sweeping and by backing up the database.

gbak

gbak is the command-line utility (found in the /bin directory of your Firebird installation) that backs up and restores databases. It is not a file-copy program: its backup operation decomposes the metadata and data and stores them separately, in a compressed binary format, in a filesystem file which, by convention, often has a filename suffix of "gbk". Restoring decompresses this file and reconstructs the database as a new database file, before feeding the data into the database objects and rebuilding the indexes.

Apart from normal data security tasks expected of a backup utility, gbak performs important roles in the regular maintenance of "database hygiene" and in the recovery of corrupted databases, provided they are backed up without garbage collection and subsequently restored without validation constraints.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Term	Definition
gdb or GDB	By convention, this is the file extension traditionally used for Firebird and InterBase databases. However, a Firebird database file can have any extension, or none at all. Many Firebird developers use ".fdb" instead, to distinguish their Firebird databases from their InterBase databases, or as part of a solution to a performance-crippling "safety feature" introduced by Microsoft Corporation into their ME and XP operating systems, targeted at files having the ".gdb" extension. Why "GDB"? It is an artifact of the name of the company that created the original InterBase, Groton Database Systems.
GDML	Mnemonic for <i>Groton Data Manipulation Language</i> , a high-level relational language similar to SQL. GDML was the original data manipulation language of InterBase, equivalent to DML in Firebird SQL but with some data definition capabilities. It is still supported through the interactive query utility, <code>qli</code> .
gdef	<code>gdef</code> is an older InterBase utility for creating and manipulating metadata. Since <code>isql</code> and the dynamic SQL interface can handle DDL, <code>gdef</code> is virtually redundant now. However, because it can output DYN language statements for several host programming languages including C and C++, Pascal, COBOL, ANSI COBOL, Fortran, BASIC, PLI and ADA, it still has its uses in the development of embedded SQL applications.
Generator	A number generating engine for producing unique numbers in series. The statement <code>CREATE GENERATOR generator_name</code> seeds a distinct series of signed 64-bit integer numbers. <code>SET GENERATOR TO n</code> sets the first number in the series. The function <code>GEN_ID(generator_name, m)</code> causes a new number to be generated, which is <i>m</i> higher than the last generated number.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Term	Definition
gfix	<p>gfix is a collection of command-line utilities, including database repair tools of limited capability. It includes tools to activate database shadows, to put the database into single-user (exclusive access) mode (shutdown) and to restore it to multi-user access (restart). It can resolve limbo transactions left behind by multiple-database transactions, set the database cache, enable or disable forced (synchronous) writes to disk, perform sweeping and set the sweep interval, switch a Firebird database from read/write to read-only or vice versa and set the database dialect.</p>
gpre	<p>In embedded database development, gpre is the preprocessor for static SQL language blocks in host language source code, translating them to BLR format in preparation for compiling. It can preprocess C, C++, COBOL and Pascal and ADA host code, on selected platforms.</p>
Grant/Revoke	<p>SQL commands GRANT and REVOKE, which are used for setting up user privileges for accessing the objects in a database.</p>
gsec	<p>gsec is Firebird's command-line security utility for managing the server-level user/password database (isc4.gdb), which applies to all users and all databases. This utility cannot be used to create or modify roles, since roles exist within a database.</p>

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

Term	Definition
gstat	<p>gstat is the command-line utility with which to gather statistics about a Firebird database. It analyzes the internal layout, like the fill factor, header page, index pages, log page and system relations. Table-by-table information about record versions (usually lengthy) can be optionally obtained, using the <code>-r</code> and <code>-t tablename</code> switches together.</p>
Hierarchical database	<p>An old design concept for implementing table-to-table relationships in databases by building up tree structures of inherited indexes.</p>
Host-language	<p>General term referring to the language in which application code is written.</p>
Identity attribute	<p>Some RDBMSs—MSSQL for example—support a table attribute which automatically implements a surrogate primary key integer column for a table and causes a new value to be generated automatically for each new row inserted. Firebird does not support this attribute directly. A similar mechanism can be achieved by explicitly defining an integer field of adequate size, creating a generator to populate it and defining a Before Insert trigger that calls the GEN_ID() function to get the next value of the generator.</p>
IBO	<p>Mnemonic for IB Objects, data access components and data-aware controls encapsulating the Firebird and InterBase API, for use with Borland Delphi and C++ Builder.</p>



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

Term	Definition
IBX	Mnemonic for InterBaseXpress, data access components encapsulating the InterBase API, distributed by Borland for use with the Enterprise editions of their Delphi and C++ Builder products.
Index	A specialized data structure, maintained by the FirebirdSQL engine, providing a compact system of data pointers to the rows of a table.
Installation	Refers to the procedure and process of copying the software to a computer and configuring it for use.
InterClient	Obsolete JDBC Type 2 Java client for the original InterBase 6 open source server. In Firebird, it is superseded by Jaybird, a Type 4 JDBC/JCA compliant Java driver developed entirely by open source developers.
InterServer	Obsolete, Java-driven server-based communication layer originally distributed with the InterBase 6 open source code release. Both InterServer and its companion InterClient have been superseded in Firebird by a more modern and open Java interface, named Jaybird.
ISC, isc, etc.	Error messages, some environment variables and many identifiers in the Firebird API have the prefix "ISC" or "isc". As a matter of purely historical interest, these initials are derived from the initial letters of "InterBase Software Corporation", the name of a subsidiary company of Borland which existed during some periods of Borland's ownership of Firebird's ancestor, InterBase.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Term

Definition

Isolation level

This attribute of a transaction prescribes the way one transaction shall interact with other transactions accessing the same database, in terms of visibility and locking behavior. Firebird supports three levels of isolation: Read Committed, Repeatable Read (also known as "Snapshot" or "Concurrency") and Snapshot Table Stability (also known as "Consistency"). Although Read Committed is the default for most relational engines, Firebird's default level is Repeatable Read, thanks to optimistic, row-level locking.

See also *Transaction Isolation*.

isql

isql is Firebird's console-mode interactive query utility, which can connect to one database at a time. It has a powerful set of commands, including a subset of Firebird SQL additional to the regular dynamic SQL command set, and a large set of embedded macros for obtaining metadata information. **isql** can output batches of commands, including embedded comments, to a file; and it can "run" such a file as a batch script—the recommended way to create and alter database objects.

JDBC

Java Database Connectivity, a set of standards for constructing database drivers for connecting Java applications to SQL databases.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Term	Definition
Join	<p>JOIN is the SQL keyword for specifying to the engine that the output of the SELECT statement involves multiple tables which can be linked by matching one or more pairs of columns specified in the subsequent ON clause.</p> <p>An INNER JOIN returns one row for each row constructed that meets all of the requirements for each table involved in the join. The keyword INNER is not required in Firebird, since INNER JOIN is the default usage of the keyword JOIN.</p> <p>An outer join returns all of the requested rows from tables designated for an OUTER JOIN, even if no matching row is found in the other “side” of the join, as follows:</p> <ul style="list-style-type: none">• in a LEFT OUTER JOIN, all requested rows from the left-hand table are returned. Where a match was made, the columns output from the right-hand table contain data. Where no match was made, these columns contain NULL.• in a RIGHT OUTER JOIN, all requested rows from the right-hand table are returned. Where a match was made, the columns output from the left-hand table contain data. Where no match was made, these columns contain NULL.• in a FULL OUTER JOIN, all requested rows from both tables are returned. Where a match was made, the columns output from both tables contain data. Where no match was made, there will be rows where either the columns from the left table or the columns from the right table all contain NULL. <p>The keyword OUTER is not required in Firebird for LEFT, RIGHT or FULL join statements.</p>
Join (inner/outer/left/right)	



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Term	Definition
jrd	The internal name of the Firebird database engine kernel. It is a mnemonic for <i>Jim's Relational Database</i> , an artifact of the original engine, invented by Jim Starkey, which became InterBase and, more latterly, Firebird.
Key	A key is a constraint on a table which is applied to a column or group of columns in the table's row structure. A key points to a unique row. A primary key or a unique key points to the unique row in which it exists; while a foreign key points to a unique row in another table by linking to its primary key or to an alternative unique key, causing dependency relationships to be created between the two tables. Primary key and unique key values must be unique in their columns throughout the table and cannot be NULL. Foreign key columns need not be unique and may be nullable under some conditions. Creating a unique index does not create a key; however, creating a primary key or unique (key) constraint creates a unique ascending index on the column(s) automatically. Creating a foreign key constraint creates an ascending non-unique index automatically. You should not create additional ascending indexes on the same columns.
Kill (shadows)	When a database shadow is created using the MANUAL keyword, and the shadow becomes unavailable, further attachments to the database are blocked. In order to re-enable attachments to the database, it is necessary to issue a <code>-kill database</code> command from the <code>gfix</code> utility to delete references to the shadow.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Term	Definition
Leaf bucket	In a binary index tree, the data item in last index on a node of the tree is known as a <i>leaf bucket</i> .
Limbo (transaction)	A limbo transaction can occur where a transaction spans more than one database. Multi-database transactions are protected by two-phase commit which guarantees that, unless the portions of the transaction residing in each database get committed, all portions will be rolled back. If one or more of the databases involved in the transaction becomes unavailable before the completion of the two-phase commit, the transaction remains unresolved and is said to be <i>in limbo</i> .
Locking conflict	Under Firebird's optimistic locking scheme, a row becomes locked against update from other transactions as soon as its transaction posts an update request for it. Where the isolation level of the transaction is SNAPSHOT TABLE STABILITY (also known as consistency), the lock occurs as soon as the transaction reads the row. A locking conflict occurs when another transaction attempts to post its own update to that row. Locking conflicts have various causes, characteristics and resolutions according to the specific settings of the transactions involved.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Term	Definition
Lock resolution	As a general term, refers to the measures taken in application code to resolve the conditions that occur when other transactions attempt to update a row that is locked by a transaction because of an update request. As a specific term, <i>lock resolution</i> refers to the WAIT/NO WAIT setting of a transaction, i.e. how a transaction should react if a locking conflict arises.
Metadata	Generally, the structure of the objects that a database comprises. Because Firebird stores the definitions of its objects right inside the database, using its own native tables, data types and triggers, the term "metadata" also refers to the data stored in these system tables.
Multi-generational architecture (MGA)	Term used for the engine mechanism of Firebird, that enables optimistic row locking and a high level of transaction isolation that protects a transaction's dynamic view of the database and its changes to rows in that view, without blocking other readers. It is achieved by the engine's capability to store multiple uncommitted versions of rows and to "age" these versions with respect to the original view, making it unnecessary to lock a row until a change is actually posted. See also <i>Versioning architecture</i> .



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Term	Definition
Natural (scan)	Seen sometimes in the query plans created by the optimizer. When preparing a query, the optimizer looks for suitable indexes to use for searching, matching, sorting, grouping and for some operations involving expressions. If it cannot find a suitable index—one which will speed up an operation without conflicting with another index—it will decide to traverse the table from the first to the last row in storage order to search for values. This is a natural scan. Since this is much slower than an indexed search, it indicates that either you have no index on the column in question or, often, you have duplicate or overlapping indexes that conflict with one another or simply confuse the optimizer. There are actually conditions where the optimizer's choice to do a natural scan is right because it has correctly determined that it would be faster than using a poor index.
Non-standard SQL	A term often heard with reference to RDBMSs with a low degree of conformance with the ISO language and syntax standards for SQL. See also <i>standard SQL</i> .
Non-unique key	This is a key that marks a column or group of columns as pointers to a matching group in another table or set. Such a key serves as a selector for a range of rows and many rows may have the same value in that key. Firebird supports only one type of formal non-unique key—the FOREIGN key constraint that can be declared to enable enforced referential integrity.

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

Term	Definition
Normalization	A technique commonly used during the analysis of data preceding database design to abstract out repeating groups of data in multiple tables and eliminate the potential to duplicate the same "facts" in related tables.
Null	<p>Sometimes wrongly referred to as a "null value", null is the state of a data item which has no known value. Logically it is interpreted as <i>unknown</i> and is thus unable to be evaluated in expressions. Notice that this is not the same as zero, blank, an empty (zero length) string, nor does it represent infinity. It represents the state of a data item which either has never been assigned a value or has been set NULL by an operation.</p> <p>Expressions and operations (arithmetic, comparison, aggregation, string concatenation, etc.) that involve NULL in an argument return a result of NULL. This can confound the relational database "newbie" who expects NULL to behave like zero, or to be ignored in aggregations.</p> <p><i>The one exception to the rule is the internal function AVG(), which does ignore null, both in aggregating the intermediate total and in counting the number of items to be aggregated.</i></p>
ODBC	Mnemonic for <i>Open Database Connectivity</i> . It is a call-level interface standard that allows applications to access data in any database which has a driver supporting this standard. Several ODBC drivers are available that support Firebird, including an open source one developed for Firebird that is compliant also with the JDBC (Java Database Connectivity) standard.

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

Term	Definition
ODS	Mnemonic for On-Disk Structure. It refers to the version of the internal structure and layout of a Firebird or InterBase database. For IB4.0, it was 8; for IB4.2, it was 8.2; for IB5.x, it was 9 and for Firebird 1 it is 10.
OLAP	A database can be raised to a higher ODS by backing it up with the gbak -r[estore] -t[ransportable] option using the old version's gbak program, and restoring from that gbak file using the new version's gbak .
Oldest active transaction	Mnemonic for <i>On-Line Analytical Processing</i> , a technology applicable when databases have grown to such a volume that it is impracticable for individuals to query them meaningfully as the basis of business decisions. Typically, OLAP systems are designed to analyze and graph trends, identify and capture historical milestones or anomalies, manufacture projections and hypothetical scenarios, crunch large volumes of data for reports, and so on, in reasonable time.
Oldest interesting transaction	Same as <i>oldest interesting transaction</i> .
	A statistic maintained by the Firebird engine, this is the transaction currently active (neither committed nor rolled back) in the database, having the lowest internal transaction ID. It can be retrieved using the -header switch of gstat command-line utility.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Term	Definition
OLE-DB	Mnemonic for <i>Object Linking and Embedding for Databases</i> . OLE is a Microsoft standard developed and promoted for incorporating binary objects of many diverse types (images, documents, etc.) into Windows applications, along with application-level linking to the software engines that create and edit them. OLE-DB was added in an attempt to provide developers with the means to supply a more vendor-specific support for database connectivity—especially relational databases—than can be achieved with “open” standards such as ODBC. More recently, Microsoft layered the ADO (Access Data Objects) technology on top of OLE-DB.
OLTP	Mnemonic for <i>On-line Transaction Processing</i> , recognized as one of the essential requirements for a database engine. Broadly speaking, OLTP refers to clients performing operations that read, alter or create data in real time.
Open source	Term used to describe software for which the source-code is available for some form of use by people other than its authors. Usually the conditions of use are described in a licensing agreement. Various public organizations have formed to formalize these conditions into recognized models offering quasi-legal protection for the authors of publicly-distributed source code. The output of these initiatives is a range of open source licensing agreements. The Firebird source code is protected by the InterBase Public License, a variant of the Mozilla Public Licence. For more information about open source licensing, see http://opensource.org and follow links.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Term	Definition
Optimization	Generally, optimization refers to techniques for making the database and application software perform as responsively as possible. As a specific term, it is often applied to the techniques used by the Firebird engine to construct efficient plans for responding to SQL requests. The code modules in the engine which calculate these plans are known collectively as <i>the Firebird optimizer</i> .
Outer join/inner join	See Join.
Page	A Firebird database is made up of fixed-sized blocks of disk space called <i>pages</i> . The Firebird engine allocates pages as it needs to. There are 10 different types of pages, all of equal size and, initially, all contain the same information. The type of page the engine stores to disk depends on the type of data object being stored on the page - data, index, blob, etc.
Page_size	The size of each fixed block is determined by the <i>page_size</i> specified for the database when the database is created. Chunks of cache memory are also allocated in <i>page_size</i> units.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Term	Definition
Parameter	A widespread term in many Firebird contexts, it can refer to <ul style="list-style-type: none">the values that are passed as arguments to and returned from stored procedures (<i>input parameters, output parameters</i>)the data items that are passed to the function blocks of the Firebird API (<i>database parameter block, transaction parameter block, service parameter block</i>)the attributes, as seen from an application development interface, of a connection instance (<i>connection parameters</i>) or a transaction (<i>transaction parameters</i>)in client applications, placeholder tokens that are accepted into WHERE and ORDER BY clauses of SQL statements, for substitution by constant values at run-time—sometimes termed <i>parameterized queries</i>
Plan	A strategy for the use of indexes for sorting and searching in queries. The Firebird optimizer always creates a plan for every query; it is also possible to supply an explicit plan by appending a PLAN clause to an SQL statement.
Platform	Term loosely used to refer to the combination of hardware and operating system software, or operating system software alone. For example, “the Windows 2000 platform”, “the Linux platform”, “UNIX platforms”. “Cross-platform” usually means “applicable to multiple platforms” or “portable to other platforms”.

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

Term	Definition
Prepare	An API function that is called before a query request is submitted for the first time, requesting validation of the statement along with several items of information about the data expected by the server.
Primary Key	A table-level constraint that marks one column or a group of columns as the key which must identify each row as unique within the table. Although a table may have more than one unique key, only one of those keys may be the primary key. When you apply the PRIMARY KEY constraint to columns in a Firebird table, uniqueness is enforced by the automatic creation of a unique ascending index named RDB\$PRIMARYn (where n is an arbitrary number appended by the Firebird engine). You should not create ascending indexes over primary key columns yourself. You should always avoid using this naming scheme for naming indexes which you have defined.
PSQL	Mnemonic for <i>Procedural SQL</i> , the subset of SQL which is available for writing stored procedures and triggers. There are minor differences between the subsets of PSQL allowed for each. For example, the context variables OLD. <i>columnidentifier</i> and NEW. <i>columnidentifier</i> are not valid in stored procedures; triggers cannot accept or return parameters nor include the EXIT statement. Unlike a stored procedure, a trigger cannot be invoked directly through a DSQLstatement, nor from a stored procedure, nor from another trigger.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Term

Definition

qli

qli (“Query Language Interpreter”) is an interactive query client tool for Firebird. It can process DDL and DML statements in both SQL and GDML, the pre-SQL query language of Firebird’s ancestor, InterBase 3. Resurfaced in Firebird, qli has some value for its capability to realize some engine features not so far implemented in Firebird SQL. Unlike its successor, isql, qli can connect to more than one database at the same time and can simulate multi-database joins.

Query

A general term for any SQL request made to the server by a client application.

RDB\$—

Prefix seen in the identifiers of many system-created objects in Firebird, a relic from Relational DataBase, the name of an early relational database developed by DEC. The design of RDB was the precursor to Firebird’s ancestor, InterBase.

RDB\$DB_KEY

The hidden internal unique key stored on every row in a table by the Firebird engine, in the past kept undisclosed to InterBase developers deliberately, because of its volatility. Because changes in row cardinality can occur without trace or warning and will always occur when a database is restored from a backup, RDB\$DB_KEY should never be treated as persistent. However, with care, it can be used dependably within an atomic operation to speed up certain operations in DSQL and PSQL dramatically.

NOTE Some middleware products—for example, IB Objects—support use of RDB\$DB_KEY and may contract its identifier to DB_KEY.

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

Term	Definition
Read Committed	The least restrictive isolation level for any Firebird transaction, Read Committed permits the transaction to update its view of the database to reflect work committed by other transactions since the transaction began. The isolation levels SNAPSHOT and SNAPSHOT TABLE STABILITY do not permit the original view to change.
Redundancy	A condition in a database where the same "fact" is stored in two unrelated places. Ideally, redundancy should be avoided by attention to normalization during analysis. However, there are circumstances where a certain amount of redundancy is justified. For example, accounting transactions often contain data items which, one could argue, could be derived by joining, looking up or calculation from other structures. However, a legal requirement to store a permanent record that will not be altered if database relationships subsequently change would overrule the case for eliminating redundancy.
Redundant Indexes	Redundant indexes often arise when existing databases are imported to Firebird from another RDBMS. When a PRIMARY KEY, UNIQUE or FOREIGN KEY constraint is applied to a column or columns, Firebird automatically creates an index to enforce the constraint. In so doing, Firebird ignores any existing indexes that duplicate its automatic index. Having duplicate indexes on keys or any other columns can defeat the query optimizer, causing it to create plans that are inherently slow.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Term	Definition
Referential integrity	<p>Generally, referential integrity is a term that refers to the way an RDBMS implements mechanisms that formally support and protect the dependencies between tables. <i>Referential integrity support</i> refers to the language and syntax elements available to provide capability to deny requests to delete or alter data objects or attributes if the action would break a dependency. It can also provide options for the actions to be taken automatically, in order for such a request to proceed.</p> <p>Firebird provides formal mechanisms for supporting referential integrity, including cascading constraints for foreign key relationships. This is sometimes referred to as <i>declarative referential integrity</i>. It also provides excellent support for user-designed referential integrity, through a very flexible implementation for defining triggers.</p>
Relation	<p>In relational database theory, a self-contained body of data formally arranged in columns and rows. In general parlance, the term is interchangeable with 'table'—except that a relation cannot have duplicate rows, whereas a table can.</p> <p>In Firebird, the terminology persists in the names of some system tables, e.g. RDB\$RELATIONS, which contains an entry for each table in the database.</p>
Relationship	<p>An abstract term referring to the way relations (or tables) are linked to one another through matching keys. Typically, for example, an Order Detail table is said to be in a <i>dependency relationship</i> or a <i>Foreign Key relationship</i> with an Order Header table.</p>



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Term	Definition
Replication	Replication is a systematic process whereby data are copied from one database to another on a regular basis, according to predictable rules, in order to bring two or more databases into a synchronized state.
Result table	A result table is the set of rows output from a SQL SELECT query. More correctly, <i>result set</i> . Synonymous with <i>output set</i> .
Roles	A standard SQL mechanism for defining a set of permissions to use objects in a database. Once a role is created, permissions are assigned to it, using GRANT statements, as if it were a user. The role is then GRANTED to individual users as if it were itself a privilege, thus simplifying the maintenance of user permissions in a database.
Rollback	In general, the act or process of undoing all of the work that has been posted during the course of a transaction. As long as a transaction has work pending that is posted but not committed, it remains unresolved and its effects are invisible to other transactions. If the client application calls for a ROLLBACK, all of the posted work is cancelled and the changes are lost. Once a transaction is committed, its work cannot be rolled back.
Schema	A term for the formal description of a database, usually realized as a script, or set of scripts, containing the SQL statements defining each and every object in the database. The term <i>schema</i> is often interchangeable with the term <i>metadata</i> .

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

Term	Definition
Schema cache	A mechanism whereby some descriptive elements of a database are stored on a client's local disk or in memory for quick reference at run-time, to avoid the need for constantly requerying the database to obtain schema (metadata) attributes.
Scrollable cursor	A cursor is a pointer to a row in a database table or output set. A cursor's position in the database is determined from the cardinality of the row to which it is currently pointing, i.e. its absolute position relative to the first row in the set. Repositioning the cursor requires returning the pointer to the first row in order to find the new position. A scrollable cursor is one which is capable of locating itself at a specified new position (upward or downward) relative to its current position. Firebird has the capability to support scrollable cursors but it has been disabled during the evolution of InterBase in the hands of Borland. At the time of writing, plans were afoot to resurface, test and, if necessary, to fulfil the implementation for a forthcoming Firebird release.
Selectivity of an index	As a general term, refers to the spread of possible values for the index column throughout the table. The fewer the possible values, the lower the selectivity. Low selectivity can also occur where an index with a higher number of possible values is represented in actual data by a very low number of actual values. Indexes of very low selectivity should be avoided, since sorts and searches based on their natural order will be faster and more efficient. A unique index has the highest possible selectivity.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Term	Definition
Server Manager	A control panel applet on a Windows Firebird server, for starting and stopping the server. In older distributions of Firebird, this applet carries the title "InterBase Manager". From Firebird 1.0 forward, several 'Firebird Manager' implementations of this applet became available.
Service Manager	In older versions of InterBase, Service Manager was a Windows-based GUI for several of the command-line utilities, whose purpose has now been superseded in Firebird by a range of excellent database administration applications.
Services API	An application programming interface to the functions of several of the Firebird command-line server utilities, providing a function-driven interface to backup, statistics, housekeeping and other utilities. At the time of writing, the Services API was applicable only to the Superserver versions of Firebird.
Sets	In relational database terms, collections of data are managed in sets consisting of one or more rows comprising one or more columns of data, each column containing one data item of a specific size and type. For example, a SELECT query specification or a view defines a set for output to a client application while an UPDATE query specification defines a set upon which the specified operation is to be performed.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Term	Definition
Shadowing and shadows	Shadowing is an optional process available on a Firebird server, whereby an exact copy of a database is maintained, warts and all, in real time, on a separate hard-disk on the same server machine where the database resides. The copy is known as a <i>database shadow</i> . The purpose is to provide a way for a database to become quickly operational after physical damage to the hard-drive where the original database resides. A shadow is not a useful substitute for either replication or backup.
Snapshot	SNAPSHOT is one of the three transaction isolation levels supported by Firebird. It provides a stable view of the database which remains current to the user of the transaction throughout the life of that transaction. It is also known as <i>concurrency level isolation</i> . See also Read Committed, Snapshot Table Stability.
Snapshot Table Stability	SNAPSHOT TABLE STABILITY is the most protective of the three transaction isolation levels supported by Firebird. It enforces a consistent view of the database for the user of the transaction, by preventing any other transaction from updating any row which it has selected, even if the transaction has not yet posted any change. It is also known as <i>consistency level isolation</i> . See also Read Committed, Snapshot.
Standard SQL	Refers to the syntax and implementation of SQL language elements as published by the International Standards Organization. This very complex standard prescribes definitions across an exhaustive range of syntax and functionality, at a number of levels.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Term	Definition
Stored Procedure	Stored procedures are compiled modules of code which are stored in the database for invocation by applications and by other server-based code modules (triggers, other procedures). They are defined to the database in a source language—procedural SQL, or PSQL—consisting of regular SQL statements as well as special SQL language extensions which supply structure, looping, conditional logic, local variables, input and output arguments, exception handling and more.
Subquery	A query specification can define output columns that are derived from expressions. A sub-query is a special kind of expression that returns a single value which is itself the output of a SELECT statement. Also known as a <i>sub-select</i> or <i>embedded query</i> .
Sub-select	See Sub-query. A <i>sub-selected column</i> is one which is specified or output from a sub-query. Such columns are not updatable.
Superserver architecture	SuperServer is the name given to the threading multi-user model that was introduced to Firebird's ancient ancestor, InterBase 4.2, to replace an older model in which each client connection was served in its own, separate process. Multiple threads share access to a single server process. The older model, which has been revived for some Firebird platforms where thread implementation is poor, is now referred to as <i>Classic</i> .



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Term**Definition****Surrogate key**

In defining a unique key, e.g. a primary key, it may occur that no column, or combination of columns, can be guaranteed to provide a unique identifier for each and every row. In that case, a column can be added and populated by values that are certain to be unique. Such a key is known as a surrogate key. In Firebird, this surrogate is most commonly defined as an integer type of suitable range and is populated automatically from a Before Insert trigger which fetches a value from a generator. In some conditions, for example where replication is to be applied, the surrogate key may be a composite of two or more such columns; or it might be a randomly-generated value such as a GUID.

Surrogate keys are also frequently used as a matter of design principle, to conform with the atomicity rules of good database design practice. See *atomic*.

Sweeping

Sweeping is the process that collects and frees obsolete versions of each record in a database when a threshold number is reached. This number, which defaults to 20,000 and is known as the *sweep interval*, is calculated on the difference between the *oldest interesting transaction* and the newest transaction. Automatic sweeping can be disabled by setting a sweep interval of zero and a manual sweep can be invoked explicitly using the `gfix` utility.

Firebird requires this sweeping because of its multiple record versioning architecture. Sweeping is not a feature of other RDBMSs which do not store obsolete record versions.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Term	Definition
System tables	Because relational database engines are self-contained, all of the metadata, or schema—data that describe the structure and attributes of database objects—are maintained within the database, in a suite of tables which is created by the CREATE DATABASE command. These tables which store “data about data” are known as system tables. Firebird system tables all have identifiers that begin with the prefix ‘RDB\$’ and actually include data about themselves, as well as every other object in the database.
Table	A term borrowed from desktop database technology, where items of data are physically stored in a tabulated format as records (rows) of fields, each record being, by definition, identical from left to right in the number and relative placement of fields and their data types and sizes. In Firebird, as in several other SQL database management systems, data are not stored in tabulated form but in contiguous blocks of disk space known as pages.
Transaction	A logical unit of work, which can involve one or many statements, which begins when the client application calls START TRANSACTION and ends when it either commits the transaction or rolls it back. A transaction is an atomic action: a commit must be able to commit every piece of pending work, otherwise all of its work is abandoned. A rollback, similarly, will cancel every piece of pending work that was posted within the transaction.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Term	Definition
Transaction isolation	A mechanism by which each transaction is provided with an environment that makes it appear (to itself) that it is running alone in the database. When multiple transactions are running concurrently, the effects of all of the other transactions are invisible to each transaction, from when it starts until it is committed. Firebird supports not just one but three levels of isolation, including one level which can see the effects of other transactions as they are committed. See <i>Read Committed</i> , also <i>Snapshot, Snapshot Table Stability</i> .
Transitively-dependent	A constraint or condition where one table (C) is dependent on another table (A), because table C is dependent on another table (B) which is dependent on A. Such a dependency would arise, for example, if table B had a foreign key referencing table A's primary key and table C had a foreign key referencing table B's primary key.
Trigger	A trigger is a module of compiled code belonging to a table, that performs an action when a DML event happens to a row in that table. Any number of event actions can be coded to occur in a prescribed sequence, before or after an insert, update or delete operation on a table's row, with virtually the full range of procedural SQL (PSQL) being available. For update triggers, the context variables NEW.ColumnName and OLD.ColumnName are available to refer to the old and new values of any column. NEW is also available to insert triggers; and OLD to delete triggers.
Tuple	In relational database terminology, the "strictly correct" name for a row in a table. Purists will say that <i>row</i> is the SQL name for a <i>tuple</i> .



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Term	Definition
UDF (“User-defined function”)	Mnemonic for User Defined Function. Firebird has but a few built-in, SQL-standard functions. To extend the range of functions available for use in expressions, Firebird can access custom functions, written in a host language such as C/C++ or Delphi, as if they were built-in. Several free libraries of ready-made UDFs exist among the Firebird community, including two that are included with the Firebird binary release distributions.
Unbalanced index	Firebird indexes are maintained as binary tree structures. These structures are said to be unbalanced when new nodes are added continually in a manner that causes major branching on one “side” of the binary tree. Typically, this will occur when a process inserts hundreds of thousands of new rows inside a single transaction. For this reason, it is recommended that indexes be deactivated during massive inserts. Subsequent re-activation will rebuild fully balanced indexes.
Uninstallation	A rather horrid word, confusing to non-English speakers, since it is not found in any self-respecting dictionary—yet! Its approximate meaning is ‘a process that is the reverse of installation’, i.e. removing a previously installed software product from a computer system.

**Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**

Term	Definition
Union	A clause in a SELECT query specification that enables the output rows of two or more SELECT statements to be combined into one final output set, as long as each of the UNIONed sets matches all of the others by the degree, data type and size of its output columns. The sets may be selected from different tables.
Updatable view	A view is said to be updatable if it is constructed from a regular query on a single table and all of its columns exist in the underlying table. Some non-updatable views can be made updatable by the creation of triggers. See also <i>View</i> .
Validation	Validation is a mechanism whereby new data applied to a column in a table are checked by some means to determine whether they fit a required format, value or range of values. Two ways to implement validation in the database are CHECK constraints and triggers. CHECK constraints will throw an exception if the input data fail to test true against an expression or constant. With triggers, the new.value can be tested more thoroughly and, if it fails, can be passed to a custom exception. The exception can be made to return the value to the client as an error, or may be handled by the trigger. For example, CHECK(Value = UPPER(Value)) may be less user-friendly than a pair of triggers (Before Insert, Before Update) that convert the new.value to upper case unconditionally.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Term	Definition
Versioning architecture	<p>Also known as <i>multi-generational architecture</i> (MGA), the feature, until recently unique to Firebird and InterBase, of placing a new version of a changed or inserted row, for the duration of a transaction, into a server-side cache where it remains visible to the transaction. When the commit occurs, the new version becomes permanent in the database and the old version is flagged as obsolete. When considering contenders for a concurrent update in a conflict situation, the engine also uses attributes of the pending record versions concerned to determine precedence, if any.</p> <p>Since the InterBase sources were released in July 2000, some other RDBMS products have begun to copy this feature.</p>
View	A view is a standard SQL object which is a stored query specification that can behave in many ways like a physical table. A view does not provide physical storage for user data: it acts as a virtual table, a predefined container for a set of output data that exist in one or more tables.
XSQLDA	Mnemonic for extended SQL descriptor area. It is a host-language data structure in the API which is used by dynamic SQL (DSQL) to transport data to or from the database when processing an SQL statement. XSQLDAs come in two types: input descriptors and output descriptors.



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Term	Definition
XSQLVAR	Structure for defining <i>sqlvar</i> , an important field in the XSQLDA structure which is used for passing and returning input and output parameters.
Y valve	Name given to the Firebird subsystem that determines which of Firebird's several "internal engines" should be used when attaching to a database. For example, one decision is whether to attach locally or as a remote client; another is to determine whether the attachment is being attempted to a database with a compatible on-disk structure (ODS).



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Alphabetical Index

Symbols

- * operator 148
- + operator 148
- / operator 148
- ; (semicolon), terminator 501
- || concatenation operator 147
- operator 148

'ALICE' 653

'BURP' 658

'DUDLEY' 665

Numerics

64-bit integer support 636

A

- Access mode 95
 - changing to read-only 474
 - changing to read-write 474

gbak

- arguments for -B 394

adding

- and subtracting date/time datatypes 251
- columns 343–344
- integrity constraints 344
- secondary files 275, 377

addition operator (+) 148

Admin Tools 12

Aggregate (function) 653

aggregate functions 143, 544

Alerter (events) 653

alerter, events 513

Aliasing

- column identifiers 141
- table identifiers 129

ALL operator 149

ALTER COLUMN 623, 630

- valid datatype conversions 348

ALTER DATABASE 377

ALTER DOMAIN 293–294

- valid datatype conversions 348

ALTER DOMAIN, available in all dialects 623

ALTER EXCEPTION 550

ALTER INDEX 359–360

ALTER PROCEDURE 529

ALTER TABLE 340–349

- arguments 349

ALTER TRIGGER 536–538

altering

- stored procedures 529

- triggers 536–538

- views 373

Alternate key 653

Alternative key 653

AND operator 150

ANY operator 149, 150

API 654

- and transactions 103

- applications 46

- BLOB control structure 602

- BLOB parameter buffer (BPB) 611

- component support 44

- Install 47

- isc_blob_ctl 602



Using Firebird

Alphabetical Index

Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- making a BLOB filter request 615
 - requesting BLOB filter in application 611
 - retrieving database statistics 464
 - Services 47
 - Services API 690
 - Apostrophes in strings 18
 - Application development 42
 - Application Programming Interface (API) 654
 - Applications
 - dynamic 43
 - embedded 43
 - multi-database 49
 - applications
 - calling stored procedures 539
 - testing 535
 - architecture, Classic vs SuperServer, compared 34
 - Argument 654
 - Arithmetic 148
 - arithmetic expressions 148–??
 - arithmetic operations, disallowed on character fields 638
 - Arithmetic operators 144, 148
 - arithmetic operators 148–??
 - precedence, listed 148
 - Array slice 654
 - array slices 266
 - Arrays
 - using stored procedures to view 545
 - arrays 232, 266–268
 - in UDFs 575
 - multi-dimensional 267
 - stored procedures and 545–548
 - subscripts 267–268
 - ASCENDING reserved word 356
 - ASCII character set 306
 - assigning values to variables 505, 506, 509
 - assignment statements 505
 - AST (asynchronous trap) function, writing 564
 - Asynchronous writes, see Forced writes 476
 - Atomicity 655
 - Atomicity of PRIMARY KEY columns 324
 - AUTO mode 384–??
 - auto vs manual shadows 384
 - AutoCommit 655
- B**
- Back up and restore
 - gbak tool 390
 - rights/permissions 392
 - why do it? 390
 - Backup 20
 - Backup and restore 390, 656
 - backing up a database to multiple files 393
 - backing up to a single file 393
 - database size and performance 402
 - examples 404
 - backup 404
 - restore 405
 - multiple-file restores 398
 - restoring on Linux/UNIX 404
 - restoring user-defined international objects 398
 - target locations for backup files 396
 - user name and password 393
 - BDE 44, 85, 656
 - single-transaction model 92



Using Firebird

Alphabetical Index

Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

BDE clients, no access to new datatypes 622
binary collation 302
Binary language representation (BLR) 658
Binary tree 657
BLOB 657
blob columns 262
BLOB control structure 586, 602, 657
blob control structure fields, listed 587
blob data, storing 262
BLOB datatype 262–265, 312
 defining ??–265
BLOB descriptors 609
 ways to populate 610
BLOB filter 658
BLOB Filters 29
BLOB filters 572, 596
 BLOB parameter buffer (BPB) 611
 custom 598
 BLOB access operations 605
 defining a column for 598
 defining the function 600
 programming filter function actions 605
 testing return value of a function 608
 writing your own 599
declaring a custom BLOB filter 598
invoking 597
making a filter request 615
pre-defined 597
read-only and write-only 600
requesting in application 611
SUB_TYPE identifiers 597
the BLOB control structure 602

blob filters 265
blob filters, declaring 596
blob relationships, illustrated 263
blob segments 262–264
blob subtypes 264–265
blob UDFs 574, 586–590
 control structures 586–588
blob_concatenate() 589
blob_get_segment 587
blob_handle 587
blob_put_segment 588
Block comments in scripts 226
BLR 658
Borland Database Engine 44, 656
Brookstone Systems InterBase Collation Kit 302
Buffer 658
buffers, database cache 219
C
C language, writing function modules 574
cache buffers 219
Cache size, setting 472
Cache, see also Buffers 659
Calculating cache size 68
calling stored procedures 539
calling UDFs 585–586
Cardinality 659
Cardinality (of a set) 120, 133
Cartesian product 132
Cascading integrity constraints 659
cascading integrity constraints 211, 213, 316, 327, 333
case sensitive names 221, 273



Using Firebird

Alphabetical Index

Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- Case-insensitive index 659
- CAST() function 254, 270
- CAST(value AS type) 145
- Casting 660
 - casting datatypes 339
 - casting date and time datatypes 634
 - changes, logging 532
- Changing ownership of a database 392
- CHAR datatype 257, 312
- CHARACTER datatype 257, 259–260
 - character repertoire, defined 301
 - CHARACTER SET 257–259, 318
- Character set 660
 - specifying in isql, scripts 185
- character set
 - default 302
 - defined 301
 - MS-DOS 307
 - names 301
 - storage requirements 304
- Character set NONE 309
- Character sets 301
 - CHARACTER SET NONE 258
 - collation order 259
 - default character set 302
 - default for a database 309
 - defaults per database element 303
 - connection 303
 - newly created database 303
 - statement 303
 - for a client attachment 310
 - for columns in a table 309
- for DOS 307
- for Microsoft Windows 308
- ISO8859_1 (LATIN_1) and WIN1252 306
- marker prefix for statements 303
 - naming of 301
 - aliases 302
- SET NAMES 310
- special 306
 - ASCII 306
 - NONE 306
 - OCTETS 306
- specifying 309
- storage requirements 304
- string literal 303
- support for Paradox and dBASE 307
- transliteration 308
 - errors 308
- transliteration errors 259
- character sets ??–312, 312–??
 - additional 308
 - choosing 209
 - corresponding to DOS code pages, listed 307
 - default 309
 - ISO8859_1 (LATIN_1) and WIN1252 306
 - NONE, OCTETS, ASCII 306
 - retrieving 308
 - specifying 275, 309–??
- character string datatypes 257–261
- character strings
 - concatenation operator (||) 147
- CHARACTER VARYING datatype 257
- characters, minimum lengths for numeric conversions



Using Firebird

Alphabetical Index

Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- 271, 294, 347
- CHECK constraints 211
 - defining ??–333
 - domains ??–291
 - triggers and 538
- Check for running server 6
- circular references 330–331
- Classic and SuperServer architectures, compared 34
- Classic architecture 661
- Classic vs Superserver 32
- client dialect
 - setting in gpre 627
 - setting in isql 626
- client version
 - compatibility issues 620
- Client-only install 16
- clients
 - and databases, migration issues 621
 - migrating 651
 - migration steps 618
 - prior versions, restrictions on using 621
- CLOB 661
- code
 - comments in 513
- code lines, terminating 501
- Code modules 494
- code pages (MS-DOS) 307
- Coercing data types 661
- COLLATE clause 145, 292, 318–319
- COLLATE sub-clause 138
- Collation Construction Kit 302
- collation names 302
- Collation order 661
 - and index key size 304
 - for a column 311
 - in comparisons 311
 - in GROUP BY 312
 - in ORDER BY 312
 - specifying 311
- collation order 259
 - and index key size 304
 - defined 209
- collation order, defined 302
- Collation orders 302
 - Firebird Collation Kit 302
 - naming of 302
- collation orders
 - and character sets, listed ??–309
 - retrieving 308
 - specifying 311–??
- Column 662
- Column aliases 141
- column defaults and column constraints 645
- COLUMN, new reserved word 628
- Columns
 - BLOB 262
 - character set 209, 318
 - CHECK constraints 211
 - COLLATE clause 318
 - collation order 209
 - COMPUTED BY 320
 - data type 208, 209
 - default values 210, 321
 - defining 315



Using Firebird

Alphabetical Index

Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- defining a CHECK constraint 331
- domain-based 319
- dropping 342
- dropping existing constraints from 345
- expression-based 320
- granting UPDATE rights for 434
- modifying 346
- NOT NULL 322
- optional attributes 316
- required attributes 316
- specifying the datatype 317
- the UNIQUE constraint 326
- UNIQUE constraints 323
- columns
 - adding 343–344
 - attributes 316–317
 - circular references 330–331
 - computed 320–??
 - datatypes 317
 - default values 321
 - defining 210, 315–333
 - domain-based 319
 - dropping 342–343, 345
 - inheritable characteristics 286
 - NULL status 211
 - NULL values 322
 - specifying character sets 309
 - specifying datatypes 317–339
 - specifying default values 210
 - specifying for views 367–??
- comments 513
- Commit 662
- COMMIT with RETAIN (COMMIT RETAINING) 102
- comparing values 504
- Comparison operators 145, 148
 - ALL 149
 - ANY 149
 - BETWEEN 149
 - CONTAINING 149
 - LIKE 149
 - SINGULAR 149
 - SOME 150
 - STARTING WITH 150
- comparison operators 148–150
 - NULL values and 150
 - precedence, listed 149
 - subqueries 148
- compatibility
 - client version 620
 - version and dialect 620
- compiled objects, and dialect 636
- compiling, UDFs 578
- composite keys 216
- Composite primary keys 324
- computed columns 320–??
- CON> prompt (isql) 159
- Concatenation of strings 18
- Concatenation operator 144, 147
- concatenation operator || 147
- Concurrency 94, 662
- Conditional joins 130
- conditions, testing 507, 508
- Configuration
 - EXTERNAL_FUNCTION_DIRECTORY 583



Using Firebird

Alphabetical Index

Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- parameters
 - cpu_affinity 63
 - deadlock_timeout 64
 - dummy_packet_interval 64
 - external_file_directory 64
 - external_function_directory 64
 - lock_acquire_spins 64
 - lock_hash_slots 65
 - server_client_mapping 65
 - server_priority_class 65
 - server_working_size_max 66
 - server_working_size_min 66
 - TCP_NO_NAGLE 65
 - tmp_directory 66
- Configuration file 62
- Configuring
 - database cache 67
 - environment variables 60
 - server names and addresses 76, 78
 - the gds_db service 78
 - the Linux/UNIX inetd daemon 79
- CONNECT statement 13
- Connecting
 - to a Linux or UNIX Server 77
 - to a Windows Server 73
- Connecting to a database 282
- connection
 - character set 303
- Connection string for TCP/IP clients
 - Windows local loopback 75
 - Windows server 75
- Connection string format 75
- connection_timeout 62
- Constants 144
- Constraint 663
- constraints
 - adding 344
 - column 645
 - declaring 333–335
 - defining 211–213, ??–333
 - dropping 345
 - table, unnamed 646
 - triggers and 538
- Contention 663
- Context variables
 - date and time 244
 - CURRENT_DATE 244
 - CURRENT_TIME 244
 - CURRENT_TIMESTAMP 244
- context variables 504
- Control Panel applet 7
- conversion
 - datatype, using ALTER COLUMN 348
- conversion functions 143
- converting datatypes 269
- converting TIMESTAMP columns to DATE or TIME 634
- Correlated subqueries 124
- Correlated subquery 663
- Corrupt database, repairing 389
- corrupt databases
 - repairing 389
- CPU affinity 63
- cpu_affinity 63
- Crash 663



Using Firebird

Alphabetical Index

Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Crash recovery 663
CREATE DATABASE 275–282
CREATE DOMAIN 286, ??–291, 319
CREATE EXCEPTION 550
CREATE GENERATOR 297–??
CREATE INDEX 354–358
CREATE PROCEDURE 495–501, ??–511
 syntax ??–525
CREATE PROCEDURE arguments 525
CREATE SHADOW 381–385
CREATE TABLE 315–333
 EXTERNAL FILE option 335–340
CREATE TRIGGER 495–501, ??–503, 532–??
 POSITION clause 535
 syntax 533–??
CREATE TRIGGER arguments 533
CREATE VIEW 366–373
Creating a database 274
creating UDFs 574
Cross join 132
Cross-database transactions 93
CURRENT_DATE function, described 244
CURRENT_DATE, new reserved word 628
CURRENT_ROLE 146
CURRENT_TIME function, described 244
CURRENT_TIME, new reserved word 628
CURRENT_TIMESTAMP 146
 available in all dialects 623
 new reserved word 628
CURRENT_TIMESTAMP function, described 244
CURRENT_USER 146
Cursors (cursor sets) 122

Cyclic links 664
D
data
 dropping 350
 exporting 340
 importing 337–339
 retrieving 506, 540
 multiple rows 526
 saving 341
 sorting 302
 updating 504
Data Definition Language 160
Data Definition Language (DDL) 199, 222
data entry, automating 532
Data Manipulation Language (DML) 222
data model 207
Data structures
 defining 199
 enforcing referential integrity 213
data structures, blob 586–588
Data Types 231
arrays 232
BLOB
 BLOB columns 262
 segment length 263
BLOB types 262
BLOBs 232
character types 257
 CHAR 257
 CHARACTER 257
 CHARACTER VARYING 257
characters vs. bytes 258



Using Firebird

Alphabetical Index

Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- collation order 259
- fixed-length character data 259
- NATIONAL CHAR 257
- NATIONAL CHAR VARYING 257
- NATIONAL CHARACTER VARYING 257
- NCHAR 257
- NCHAR VARYING 257
- specifying a character set 257
- transliteration errors 259
- using CHARACTER SET NONE 258
- VARCHAR 257
- variable-length character data 261
- date and time types 243
 - "sliding century window" 250
 - casting between 254
 - converting to 253
 - DATE 243
 - operations using 251
 - TIME 243, 249
 - TIMESTAMP 243
- exponential notation 236
- fixed-decimal (scaled) 233
 - DECIMAL 238
 - how they are stored 234
 - NUMERIC 237
 - operations using 239
 - precision and scale 235
 - specifying in embedded applications 241
 - using in arithmetic 238
- floating-point 242
 - DOUBLE PRECISION 242
 - FLOAT (single precision) 242
- precision 242
- integer types
 - 64-bit (NUMERIC(18,0)) 236
 - INT64 236
 - INTEGER 236
 - SMALLINT 236
- supported types 231
- where to specify 232
- Data types
 - array types 266
 - defining 266
 - element data types 266
 - multi-dimensional 267
 - when to use 266
 - BLOB
 - BLOB filters 265
 - subtypes 264
 - viewing 266
 - coercing 661
 - converting 269
 - CAST() function 270
 - changing column and domain definitions 270
 - explicit type conversions 270
 - implicit type conversions 269
- Database 664
 - cache 67
 - calculating size 68
 - changing size 69
 - Classic server default 68
 - Superserver default 68
 - connecting to 12, 282
 - employee.gdb 12



Using Firebird

Alphabetical Index

Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- path 13
- database
 - create using isql 15
 - on a CD-ROM 4
 - read-only 4
- database and server migration steps 618
- database cache buffers 219
- database dialect
 - setting in gfix 627
- Database events
 - asynchronous signaling 563
 - handling on a client 560
 - listening with EVENT WAIT 564
 - multiple 563
 - POST_EVENT 560
 - registering interest in 562
 - responding to 565
 - writing an asynchronous trap (AST) function 564
- database objects 201
- Database repair 480
- Database repair How-to 481
- Database shadowing 379
 - auto mode and manual mode 384
 - conditional shadows 385
 - dropping a shadow 385
 - multi-file 383
- Database statistics 452
- Database validation 480
- Database-level security 429
- Databases 272
 - 'Single-user mode' 283
 - ALTER DATABASE 377
- altering 377
- analyzing requirements 203
- background garbage collection 386
- backup and restore 390
- changing dialect 474
- collecting and analyzing data 203
- CREATE DATABASE 275
- CREATE SHADOW 381
- creating 274
 - specifying user name and password 279
- creating a multi-file shadow 383
- data model database 201
- database shadows 379
- description and analysis 200
- design framework 202
- design goals 201
- designing 199
- Dialect 2 475
- Dialect 3 475
- dropping 379
- enforcing referential integrity 213
- events 512
- file-naming conventions 274
- forced writes 388
- garbage collection 386
- garbage collection during backup 387
- getting exclusive access to 283
- identifying entities and attributes 204
- increasing cache size 219
- making a database read-only 376
- multi-file 277
- naming objects 220



Using Firebird

Alphabetical Index

Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- normalizing 214
- object naming conventions 273
- planning security 220
- quote-delimited identifiers 273
- read-only 375
- relationships between objects 212
- repair How-to 481
- repairing a corrupt database 389
- requirements analysis 203
- restoring multiple-user access 284
- secondary files 377
- shadow location 382
- shadowing 379, 488
 - auto mode and manual mode 384
 - conditional shadows 385
 - dropping a shadow 385
 - to drop a shadow 488
 - to activate a shadow 488
- single-file 276
- specifying the default character set 281
- statistics 452
- sweeping 387
- the database cache 388
- Use All Space 475
- using CHARACTER SET NONE 281
- validation and repair 389, 480
 - when to increase page size 280
- databases
 - and clients, migration issues 621
 - and servers, migrating 638
 - dialect 1 641
 - multifile 278–280
- normalization 200, 214–218
- older, migrating 650
- page size 272
 - changing 275, 280
 - default 280
 - overriding 280
 - read-only 284, 376
 - repairing 484
 - shadowing 379–386
 - single-file 276–277
 - structure 199, 201
 - sweeping ??–487
 - sweep interval 488
 - validating 389, ??–481
- datatype conversion
 - using ALTER TABLE ALTER COLUMN 348
- datatypes 231, 231–270
 - casting 339
 - converting 269
 - domains 287–288
 - DSQL applications 241
 - precision 241
 - specifying 232–233
 - specifying for columns 317–339
 - stored procedures and 505
 - XSQLVAR field 241
- datatypes for UDF parameters 574
- date and time datatypes, casting 634
- date and time value functions 244
- DATE datatype 243, 244
 - available only in dialect 3 624
 - converting from TIMESTAMP 634



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- dialect 1 247
- dialect 3 247
- interpreted as TIMESTAMP by older clients 621
- older clients cannot access 621
- SQL 92 compliant 625
- transition feature 619
- Date literals 145, 245
 - predefined 249
 - 'NOW' 249
 - 'TODAY' 249
 - 'YESTERDAY' 249
 - recognized formats for 245
- date literals 245–249
 - defined 245
 - predefined 249
- date string separators 248
- DATE, TIME, and TIMESTAMP datatypes
 - migrating 630–635
 - transition feature 619
- DAY, new reserved word 628
- DB_KEY see RDB\$._KEY 566
- dBASE
 - collation names 302
- dBASE collations 307
- Security
 - the gsec utility
 - interactive commands
 - a 424
- DDL 160, 664
- DDL (Data Definition Language) 116, 160, 222
- DDL (Data Definition Language) 199
- DDL and DML statements, in client dialect 1 629
- DDL script 160
- DDL Scripts 222
- Deadlock 101, 664
- Deadlock timeout parameter 64
- deadlock_timeout 64
- debugging stored procedures 513
- DECIMAL and NUMERIC datatypes, migrating 635
- DECIMAL datatype 237–241
- DECIMAL datatype, transition feature 619
- DECLARE EXTERNAL FUNCTION arguments, listed 581
- declaring
 - input parameters 503
 - integrity constraints 333–335
 - output parameters 504
- declaring blob filters 596
- Declaring the PRIMARY KEY 325
- Default cache size, setting 472
- default character set 302, 309
- DEFAULT column value 210
- default values
 - column 321
 - specifying 210
- defining
 - column, default value 210
 - columns 210, 315–333
 - integrity constraints 211–213, ??–333
- Degree (of a set) 120
- Degree (of an output set) 664
- DELETE statements
 - calling UDFs 586
- DELETE, triggers and 532



Using Firebird

Alphabetical Index

Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- delimited identifier, enclosed in double quotes 273
- Delimited identifiers 273
- delimited identifiers 221, 629
 - available only in dialect 3 623
 - older clients cannot access 622
 - SQL 92 compliant 625
- DEPARTMENT table 215
- Deployment 664
- DEPT_LOCATIONS table 216
- DESCENDING reserved word 356
- designing
 - tables 207
- dialect
 - and compiled objects 636
 - client, setting in gpre 627
 - client, setting in isql 626
 - creating databases with isql 626
 - database, setting in gfix 627
 - defined 618
- dialect 1
 - clients 624
 - clients, DDL vs DML statements 629
 - databases 624, 641
 - DOUBLE PRECISION data storage 624
- dialect 1 DATE datatype 247
- dialect 2, for detecting migration problems 625
- Dialect 3 623
- dialect 3
 - clients and databases 625
 - exclusive features 623
- dialect 3 DATE datatype 247
- Dialect, to change 474
- dialects 618
 - features available in all 623
 - using, in InterBase tools 625
- Security
 - the gsec utility
 - interactive commands
 - mo 425
- Directory for external files 64
- Dirty reads 91
- dirty reads 92
- division operator (/) 148
- DLLs and UDFs 573–579
- DML 664
 - DML (Data Manipulation Language) 222
 - DML scripts 222
- Domain 665
- domain-based columns 319
- Domains 285
 - ALTER DOMAIN 293
 - altering 293
 - CHARSET/CHARACTER SET attribute 292
 - CHECK data conditions 290
 - COLLATE attribute 292
 - CREATE DOMAIN 286
 - data type for 287
 - DEFAULT attribute 288
 - default values 210
 - domain identifier 286
 - dropping 295
 - NOT NULL attribute 289
 - overriding 286
 - specifying 210



Using Firebird

Alphabetical Index

Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

domains 210, ??–295
 altering 293–294
 attributes 287
 creating ??–291
 datatypes 287–288
 dropping 295
 specifying defaults ??–289
DOUBLE PRECISION data storage in dialect 1 624
DOUBLE PRECISION datatype 242–243
double quotes 623
 disallowed in client DDL statements 630
 string constants in dialect 1 DML 629
 transition feature 619
 use only for delimited identifiers 623
 vs single quotes 623, 629
double quotes, enclosing delimited identifier 273
double-quote delimited identifiers 221
Double-quoted identifiers 17
DPB 665
DROP DATABASE 379
DROP DOMAIN 295
DROP EXCEPTION 550
DROP INDEX 361
DROP PROCEDURE 530
DROP SHADOW 385
DROP TABLE 349–??
DROP TRIGGER 538
dropping
 columns 342–343, 345
 constraints 345
 data 350
 views 374

DSQL 43, 665
 applications, datatypes 241
 stored procedures and 539
DSQL (dynamic SQL) 113
Duplicate rows, preventing 355
duplicating triggers 535
dyn, also DYN 665
Dynamic applications 43
Dynamic SQL 43
E
Security
 the gsec utility
 interactive commands
 h 426
Embedded applications 43
Embedded query 692
Embedded SQL 112
EMPLOYEE table 207, 212, 218
END 509–511
entities 201, 204, 207
entities and attributes, listed 205
Environment variables 60
 INTERBASE 61
 INTERBASE_LOCK 61
 INTERBASE_MSG 61
 INTERBASE_TMP 61
 ISC_PASSWORD 60
 ISC_USER 60
 TMP 61
Error 666
Error Code 666
error codes 554



Using Firebird

Alphabetical Index

Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- error messages 549
- error-handling routines
 - stored procedures 552–??
- errors 554
 - stored procedures 509, 511, 555
 - triggers 534, 549
- ESQL 666
 - ESQL (embedded SQL) 112
 - and -R 399
 - Event 666
 - Event alerter 653
 - Events 512
 - asynchronous signaling 563
 - handling on a client 560
 - listening with EVENT WAIT 564
 - multiple 563
 - POST_EVENT 560
 - registering interest in 562
 - responding to 565
 - writing an asynchronous trap (AST) function 564
 - events 513
 - exact numerics, transition feature 619
 - exact numerics: INTEGER, SMALLINT 236
 - EXCEPTION 551
 - Exception 666
 - Exceptions
 - ALTER EXCEPTION 550
 - CREATE EXCEPTION 550
 - DROP EXCEPTION 550
 - user-defined 550
 - exceptions 532, 549–551
 - dropping 550
- handling 552
- Exclusive access to database 283
- executable procedures 539
 - terminating 509
- Execute 667
 - EXECUTE PROCEDURE 526, 539
- EXISTS operator 149
 - EXISTS() predicate 127, 149
- EXIT 509–511
 - Explicit join syntax 130
- exporting data 340
- expression elements, listed 144
- expression-based columns See computed columns
- Expressions
 - as column definitions for views 368
 - involving NULL 19
- Expressions (SQL) 143
- External file
 - exporting to 340
- external file format 652
- EXTERNAL FILE option 335–340
 - restrictions 336–337
- external files 335
- External files as tables 335
- External function directory 64
 - external_file_directory 64
- EXTRACT(), available in all dialects 623
- EXTRACT, new reserved word 628
- Extracting metadata 193
- F**
- factorials 527
- features available



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

- in all dialects 623
- only in dialect 3 623
- FIBPlus 45, 667
- file specification: syntax, by platform 276
- files
 - exporting 340
 - external 335
 - importing 337–339
 - primary 276–277
 - secondary 275, 277–280, 377
- Firebird
 - API 42
 - client 39
 - gds32.dll (Windows) 41
 - gdslib.so (Linux/Unix) 41
 - configuration file 62
 - header file (ibase.h) 103
 - origins 25
 - queries 110
 - SQL 110
 - summary of features 27
- Firebird client 26
- Firebird Manager applet 8
- Firebird Project 23
- Firebird server 41
 - resources required 26
 - supported platforms 26
- Firebird Service Manager 402
 - backing up with 403
 - for restoring 403
- Firewalls 81
- firing triggers 534
- security 549
- fixed-decimal datatypes 237–241
- FLOAT datatype 242–243
- floating-point datatypes 242–243
- FOR SELECT . . . DO 526
- Forced writes 21, 388, 476
 - do not disable on Windows server 21
 - enabling and disabling 476
- Foreign Key 667
- FOREIGN KEY constraints 211–213, 326, 326–329, 354
- FREE_IT 577
- Full join 131
- Full outer join 131
- functions
 - aggregate 143
 - conversion 143
 - numeric 143
- G**
- Garbage collection 485, 668
- gbak 668
 - backing up a database to multiple files 393
 - backing up to a single file 393
 - changing ownership of a database 392
 - database size and performance 402
 - error messages 406
 - examples 404
 - backup 404
 - restore 405
 - logging in during restore 22
 - multiple-file restores 398
 - restoring on Linux/UNIX 404



Using Firebird

Alphabetical Index

Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- restoring to a running database 22
- restoring user-defined international objects 398
- return value (Windows only) 396, 401
- running a restore 397
- security considerations
 - Backup and restore
 - security considerations 397
 - target locations for backup files 396
 - the -service switch 402
 - for backups 403
 - for restoring 403
 - transportable backups 397
 - upgrading the on-disk structure 391
 - user name and password 393
 - using gbak with Firebird Service Manager 402
 - backing up 403
 - for restoring 403
- gbak tool 390
- gdb or GDB 669
- gdef 669
- GDML 669
- GDML, migrating databases created with 650
- gds_db service 78, 83
- gds_lock_pr 467
- GEN_ID() 512
- Generator 669
- Generators 285, 296
 - 64-bit 638
 - and data types 297
 - creating 297
 - deleting 300
 - DROP GENERATOR 300
- fetching the previous value 298
- GEN_ID() function 300
- how they are used 296
- in stored procedures and triggers 512
- retrieving a value into an application 299
- SET GENERATOR 299
- setting or resetting value 299
- using 298
- using GEN_ID() to decrement 299
- vs autoincrement types 296
- generators 512
 - defined 296
 - resetting, caution 300
- generators, recreating 638
- Get help 23
- gfix 670
 - activating a shadow 488
 - changing database settings 472
 - access mode 474
 - database dialect 474
 - enabling/disabling 'Use All Space' 475
 - enabling/disabling Forced Writes 476
 - read-only ~ read-write 474
 - changing database settings setting default cache size 472
 - database repair 480
 - database validation 480
 - dropping a shadow 488
 - error messages 492
 - version of Firebird server installed 477
 - getting database access 470
 - gfix -shut (shutdown) 471



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- kill shadow databases 488
- password switch for 470
- recovery of limbo transactions 477
- shadows 488
 - shutting down a database 471
 - terminating a shutdown 472
 - to disable automatic sweeping 486
 - to perform a manual sweep 487
 - when to validate a database 480
- gfix: setting database dialect 627
- gpre 670
 - dialect precedence 627
 - setting client dialect 627
- gpre: blob data 264
- GRANT 429–??
 - multiple privileges 435–436
 - multiple users 437
 - privileges to roles 432
 - specific columns 434
 - TO TRIGGER clause 549
 - WITH GRANT OPTION 440–??
- grant authority
 - revoking 451
- Grant/Revoke 670
- GROUP BY
 - collation order 312
- gsec 670
- gstat 671
- gstat command-line tool 452
- Guardian 6
- H**
- Handle
- transaction 104
- HAVING sub-clause 137
- headers
 - procedures 504
 - triggers 532, 534–535
 - changing 537
- Help 23
- Hierarchical database 671
- Host-language 671
- Host-language variables 144
- host-language variables 506
- HOSTS file 76
- HOUR, new reserved word 628
- Housekeeping utilities 469
 - gfix 469

I

- IB Objects 45
- ibase.h 103
- ibconfig 62
- iblockpr 467
- IBO (IB Objects) 671
- ibserver process 83, 84
- IBX (InterBaseXpress) 672
- identifiers
 - case sensitive 221
 - delimited 221
- identifiers, using reserved words for 622
- Identity attribute 671
- IF . . . THEN . . . ELSE 508
- Implicit join syntax 130
- importing data 337–339
- IN clauses, limit 638



Using Firebird

Alphabetical Index

Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- IN() predicate 128
- in() predicate 149
- Inclusive vs exclusive OR 151
- incorrect values 545
- incremental values 512
- Index 672
 - index key size, and collation order 304
 - index tree 272
- Indexes
 - ALTER INDEX 359
 - automatic 354
 - on foreign key columns 354
 - on primary key columns 354
 - on UNIQUE-constrained columns 355
 - basics 352
 - deactivating for batch inserts 360
 - DROP INDEX 361
 - duplicating system indexes 355
 - improving performance 359
 - backup and restore 359
 - housekeeping 359
 - rebuilding an index 359
 - recomputing index selectivity 359
 - inspecting 354
 - on existing columns containing duplicates 356
 - preventing duplicate entries 355
 - recompute the selectivity of an index 360
 - redundant 686
 - selectivity 361
 - specifying sort order 356
 - to optimize OR queries 357
 - to rebuild 359
- using gbk for index cleanup 362
- using SET STATISTICS 360
- when to index 353
- when to use a multi-column index 357
- indexes 219
 - activating/deactivating 359
 - altering 359–361
 - creating 354–358
 - defined ??–353
 - dropping 361
 - improving performance 359–??
 - multi-column 352, 355, ??–358
 - page size 219
 - preventing duplicate entries 355
 - rebalancing 359
 - rebuilding 359
 - recomputing selectivity 360
 - single-column 352, 355
 - sort order 355, 356
 - system level, automatically generated 354
 - system-defined 354, 361
- inetd 79
- Inxes
 - creation of 354
- in-place migration 642
- input parameters 503
- INSERT, calling UDFs 585
- INSERT, triggers and 504, 532
- inserting unique column values 512
- Install API 47
- Installation 672
 - disk locations 1



[Symbols](#) [Numerics](#) [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

- Firebird 1.5 onward 1
- prior to Firebird 1.5.1
- drives 4
- kits 1
- script or program 4
- INT64**
 - available only in dialect 3 624
 - columns, older clients cannot access 621
 - data storage
 - for DECIMAL and NUMERIC datatypes 619
 - for large exact numerics 624
 - only in dialect 3 columns 625
- INT64 type** 636
- integer datatypes 236–237
- integrity constraints
 - adding 344
 - cascading 211, 213, 316, 327, 333
 - declaring 333–335
 - defining 211–213, ??–333
 - dropping 345
 - on columns 316
 - referential integrity check options 327
 - triggers and 538
- interactive SQL Utility (isql) 152
- InterBaseXpress (IBX) 45
- InterClient 46, 672
- Interleaved transactions 92
- Internal context variables 144
- international character sets 209, ??–312, 312–??
 - additional 308
 - default 309
 - specifying 309–??
- InterServer 672
- ISC, isc, etc. 672
- isc_blob_ctl 602
- isc_config 62
- isc_decode_date() 254
- isc_encode_date() 254
- ISC_PASSWORD 153, 393, 471
- isc_start_multiple() 105
- isc_start_transaction() 105
- ISC_USER 153, 393, 471
- ISO8859_1 character set 306
- Isolation level 94, 96, 673
- isql 274, 673
 - command-line switches 195
 - connecting to a database 154
 - from remote workstation 154
 - locally 154
 - creating/altering database objects 159
 - database dialect, setting 626
 - default dialect 626
 - dialect precedence 626
 - ending an interactive session 195
 - error handling 156
 - extract metadata for in-place migration 642
 - extracting metadata 193
 - general commands 160
 - BLOBDUMP 161
 - EDIT 161
 - EXIT 162
 - HELP 162
 - INPUT 163
 - OUTPUT 164



Using Firebird

Alphabetical Index

Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- QUIT 164
- SHELL 165
- interactive commands 158
- invoking 153
- modes for use 152
- modes for use
 - command-line 153
 - interactive 152
- SET commands 180
 - SET AUTO 180, 181
 - SET AUTODDL 180
 - SET BLOBDISPLAY 181
 - SET COUNT 182
 - SET ECHO 183
 - SET NAMES 185
 - SET PLAN 186
 - SET PLANONLY 187
 - SET SQL DIALECT 187
 - SET STATS 188
 - SET TERM 189
 - SET TIME 190
 - SET WARNINGS 191
- setting client dialect 626
- setting dialect in/with 157
- SHOW commands 165
 - SHOW CHECK 166
 - SHOW DATABASE 166
 - SHOW GRANT 171
 - SHOW INDEX (SHOW INDICES) 172
 - SHOW SQL DIALECT 175
 - SHOW SYSTEM 176
 - SHOW TABLE(S) 176
- SHOW VERSION 179
- starting 15
- starting an interactive session 159
- stored procedures and 539–545
- terminator character 154
- TIMESTAMP display 638
- triggers and 532–??
- using dialect 2, for metadata problems 625
- using warnings 156
- ISQL statements, terminating 501
- isql utility 152
- J**
 - Java Type 4 JDBC driver 46
 - Jaybird 46
 - Jbird 46
 - JDBC 673
 - Join 674
 - Join (inner/outer/left/right) 674
 - JOIN operations 129
 - ambiguity 129
 - joins and views 365
 - jrd 675
- K**
 - Key 675
 - Keys
 - alternative 653
 - atomic 324
 - composite 324
 - determining unique attributes for 208
 - non-unique 678
 - primary 208
 - surrogate 693



Using Firebird

Alphabetical Index

Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- unique 208
- keys
 - composite 216
 - removing dependencies 216–217
- keywords Seereserved words
- Kill (shadows) 675
- L**
 - large exact numerics
 - data storage and precision 624
 - transition feature 619
 - LATIN_1 character set 306
 - Leaf bucket 676
 - Left join 131
 - Left outer join 131
 - legacy databases, migrating 650
 - Security
 - the gsec utility
 - interactive commands
 - de 425
 - Limbo (transaction) 676
 - Limbo transactions 94
 - Limbo transactions recovery 477
 - Livelock 102
 - Local access (Windows) 72
 - local columns 287
 - Local connection (Windows)
 - connection string 73
 - server_client_mapping 65
 - Local loopback 77, 82
 - local variables 502, 503
 - assigning values to 505
 - Lock Print utilities 467
 - Lock resolution 96, 677
 - Locking and lock conflicts 100
 - Locking conflict 676
 - logging changes 532
 - Logging in during restore 22
 - Logical operators 144
 - AND 150
 - NOT 150
 - OR 150
 - logical operators
 - precedence, listed 150
 - Lost updates 91
 - M**
 - Managing security 30, 414
 - MANUAL mode 384–385
 - masterkey password 11
 - mathematical operators 148–??
 - precedence 148
 - max_seglen 588
 - memory, allocating for UDFs 576
 - Metadata 199, 222, 677
 - metadata
 - double quotes disallowed 630
 - extracting, with isql 642
 - legacy, use dialect 2 625
 - storing 199
 - Microsoft C compiler options 578
 - migrating
 - clients 651
 - data from other DBMS products 652
 - dates and times 630–635
 - DDL and DML statements 629



Using Firebird

Alphabetical Index

Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- generators 638
- NUMERIC and DECIMAL datatypes 635, 647, 648
- older databases 650
- servers and databases 638
 - to a new database 649
 - to a new server 640
 - using external file format 652
- Migrating to Firebird
 - "in-place" server migration 639
 - 64-bit integer support 636
 - ambiguous column correlations 619
 - bug-fixes that may impact you 617
 - checklists
 - clients 618
 - databases 618
 - clients 651
 - clients and database 621
 - connection path strings 619
 - data types 617
 - delimited (double-quoted) identifiers 617
 - dialect
 - table of function by dialect 620
 - dialects 617, 618, 623
 - about dialect 1 databases 641
 - behavior of compiled objects 636
 - date and time types 630
 - the EXTRACT() function 633
 - decimal and numeric types 635
 - dialect 1 clients and databases 624
 - dialect 2 clients 625
 - dialect 3 only 623
 - generators 638
- miscellaneous issues 638
- scaled numeric types 635
- transforming databases to dialect 3 642
- migrating servers and databases 638
- migrating to a new server 640
- on-disk structure 620
- reserved words used as identifiers 622
- servers and databases 621
- setting SQL client dialect
 - double quotes 629
 - gpre 627
 - new reserved words 628
 - single quotes vs double quotes 629
- setting SQL database dialect
 - gfix 627
- setting SQL dialect 625
- SQL dialects 623
- transition features regarding data types 619
- migration issues 618–622
- migration problems, and dialect 2 625
- MINUTE, new reserved word 628
- MONTH, new reserved word 628
- MS-DOS code pages 307
- multi-column indexes 352, ??–358
 - defined 355
- Multi-database applications 49
- Multi-file database
 - creating 278
 - size parameters for multiple files 278
 - specifying a secondary file using LENGTH 278
 - specifying the starting page number of a secondary file 279



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- multifile databases 278–280
- multifile shadows 383
- Multigenerational architecture 29
- Multi-generational architecture (MGA) 677
- Multiple CPUs 63
- Multiple row output from stored procedures 526
- Multiple transactions 93
- multiple triggers 535
- multiplication operator (*) 148
- Muti-file databases 277
- N**
 - Named constraints 333
 - names
 - of character sets 301
 - names, case sensitive 221, 273
 - naming
 - objects 220
 - triggers 534
 - variables 513
 - naming database objects 274
 - NATIONAL CHAR datatype 257, 259–260
 - NATIONAL CHAR VARYING datatype 257
 - Natural (scan) 678
 - Natural join 132
 - NCHAR datatype 257, 261
 - NCHAR VARYING datatype 257
 - Nested sets 123
 - nested stored procedures 527–529
 - NETBEUI 72, 74
 - Network
 - protocols 72
 - Local access (Windows) 72
 - Mixed platforms 72
 - NetBEUI 72
 - TCP/IP 72
 - NEW and OLD context variables (trigger language) 504
 - NEW context variables 504
 - NO RECORD_VERSION 98
 - NONE character set 306
 - NONE reserved word 258–259, 281
 - Non-reproducible reads 91
 - Non-standard SQL 678
 - Non-unique key 678
 - Normalization 679
 - normalization 200, 214–218
 - NOT operator 150
 - NOW date literal 249
 - NOWAIT 96
 - NULL 19, 149, 679
 - Null 679
 - NULL and user-defined functions 126
 - NULL in calculations 126
 - NULL in expressions 125
 - NULL status 211
 - NULL values
 - columns 322
 - NULL values, comparisons 150
 - number_segments 587
 - numbers, incrementing 512
 - NUMERIC and DECIMAL datatypes, migrating 647, 648
 - NUMERIC and DECIMAL datatypes: storage 234
 - NUMERIC datatype 237–241
 - NUMERIC datatype, transition feature 619



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

numeric datatypes ??–243

numeric functions 143

0

Object is in use (error) 518

object names, case sensitive 273

Object naming conventions 273

objects 201

- relationships 212

- valid names for 220

OCTETS character set 306

ODBC 679

ODS 680

ODS (on-disk structure) 620

ODS data format 621

- compatibility issues 620

OLAP 680

OLD context variables 504

Oldest (interesting) transaction 106

Oldest active transaction 106, 680

Oldest interesting transaction 680

OLE-DB 681

OLTP 681

ON DELETE 213, 327

ON UPDATE 213, 327

On-disk structure 680

- to upgrade 391

on-disk structure 619

On-disk structure (ODS) 620

Open source 681

operator type precedence 147

operators

- arithmetic 148–??

comparison 148–??

Operators (SQL) 143

Optimistic row-level locking 29

Optimization 682

optimizing queries 357

OR operator 150, 151

ORDER BY

- collation order 301, 312

ORDER BY clause 138, 357

order of evaluation (operators) 147–??

Outer join 131

output 539

output parameters 504, 509

- viewing 539

P

Page 682

page size 272

- indexes 219

- shadowing 384

PAGE_SIZE 67

Page_size 682

Paradox

- collation names 302

Paradox collations 307

Paradox for DOS 307

Paradoxcharacter sets 308

Parameter 683

parameters

- input 503

- output 504, 509

- viewing 539

parameters, UDFs 574



Using Firebird

Alphabetical Index

Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- Parentheses 146
partial key dependencies, removing 216–217
password
 default 11
 to change 11
passwords
 specifying 275, 279–280
Path 13
Path strings
 inconsistent 76
Permissions
 for backup and restore 392
Phantom rows 92
Ping 5, 80
Plan 683
Platform 683
Port 3050 83
precedence of operator types, listed 147
Precision 235
precision
 DECIMAL and NUMERIC datatypes 619
 large exact numerics 624
precision of datatypes 241
Predefined date literals 249
predefined date literals 249
Predicates 124
elements 124
EXISTS() 127, 149
IN() 128, 149
NULL 125
NULL in expressions 125
operators 124
values 125
Prepare 684
primary files 276–277
Primary Key 684
PRIMARY KEY constraints 208, 211–213, 323–326, 354
Procedural SQL language (PSQL) 494
Programming
 on server 48
Programming on Firebird server 494
Programming on server
 PSQL (procedural SQL language) 494
PROJECT table 212, 216, 217
PSQL 684
 extensions for stored procedures 497
PSQL (procedural SQL language)
 overview of code modules 494
PUBLIC reserved word 437
Q
qli 685
Queries 116
 optimizing with RDB\$DB_KEY 568
queries
 optimizing 357
 search conditions
 combining simple 150
 reversing 150
Queries that count rows 133
Query 685
Query plan
 view without executing query 187
quote marks



Using Firebird

Alphabetical Index

Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- double vs single 629
- inside strings 643
- Quote-delimited identifiers 273
- R**
 - raising exceptions 551
 - RDB\$— 685
 - RDB\$_KEY 566
 - RDB\$DB_KEY 685
 - duration of validity 570
 - qualifying in an output set 571
 - using to optimize queries 568
 - RDB\$RELATION_CONSTRAINTS system table 334
 - READ COMMITTED 98
 - Read Committed 686
 - read-only database 4
 - read-only databases 284, 376
 - read-only databases, available in all dialects 623
 - read-only views 368–370
 - gbak
 - arguments for -C 399
 - Record versions 94, 95
 - RECORD_VERSION 98
 - RECREATE PROCEDURE 514
 - recursive stored procedures 527–529
 - Redundancy 686
 - Redundant indexes 686
 - REFERENCES privilege 329, 434
 - Referential integrity 323, 327, 659, 687
 - circular references 330
 - optional referential trigger actions 327
 - CASCADE 327
 - NO ACTION 327
 - SET DEFAULT 328
 - SET NULL 328
 - orphan rows 330
 - referring to tables owned by others 329
 - Relation 687
 - relational model 212
 - Relationship 687
 - relationships between objects, illustrated 204
 - Repair utility 469
 - gfix 469
 - repairing databases 484
 - repeating groups, eliminating 214–216
 - repetitive statements 507, 526
 - Replication 688
 - requirements, analyzing 203
 - Reserved words
 - new in Firebird 628
 - reserved words
 - new 628
 - restrictions on using 628
 - used as identifiers 622
 - Restore
 - see Backup and restore 398
 - see gbak 398
 - Restoring to a running database 22
 - Result table 688
 - retrieving data 506, 540
 - multiple rows 526
 - return values
 - UDFs 575
 - return values, stored procedures 504
 - incorrect 545



Using Firebird

Alphabetical Index

Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

REVOKE 446–??

grant authority 451

multiple privileges 448–451

multiple users 449

restrictions 448

stored procedures 450

Right join 132

Right outer join 132

Roles 688

roles 438, 449

granting 431, 432

granting privileges to 438

granting to users 439

revoking 449

Rollback 688

routines 532

rows

retrieving 506, 540

multiple 526

S

isql

SHOW commands

SHOW DOMAIN 167

SHOW EXCEPTION 168

SHOW FUNCTION 169

SHOW GENERATOR 170

SHOW PROCEDURE 173

SHOW ROLE 175

SHOW TABLE 176

SHOW TRIGGER 177

SHOW VIEW 180

Scale 235

Scecurity

UDF libraries 583

Schema 222, 688

Schema cache 689

Scripts 222

/* */ comment markers 226

basic steps 224

chaining 222

comments 226

-- comment marker 227

block comments 226

in-line comments 226

one-line comments 227

committing statements in 230

creating 225

DML statements in 230

executing 224

how to create 223

isql statements in 227

linking multiple 222

SET TERM statements 227

SQL statements 226

terminator symbols 227

terminators and procedure language (PSQL) 228

what is in them? 226

why use them? 223

Scrollable cursor 689

search conditions (queries)

reversing 150

simple, combining 150

SECOND, new reserved word 628

secondary files 277–280



Using Firebird

Alphabetical Index

Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- adding 275, 377
- Security 414
 - backup and restore 397
 - database-level 429
 - ALL privileges 430
 - default security and access 429
 - DELETE privileges 430
 - effects of DML operations on updatable views 445
 - EXECUTE privileges 430
 - granting privileges 432
 - granting privileges for a stored procedure or trigger 434
 - granting privileges for views 443
 - granting privileges through roles 438
 - granting privileges to a list of procedures 437
 - granting privileges to a UNIX group 437
 - granting privileges to all users 437
 - granting privileges to execute stored procedures 442
 - granting UPDATE rights for columns 434
 - granting users the right to grant privileges 440
 - implementing roles 431
 - INSERTprivileges 430
 - multiple privileges and multiple grantees 435
 - REFERENCES privileges 430
 - revoking a role from users 449
 - revoking all privileges 448
 - revoking EXECUTE privileges 450
 - revoking grant authority 451
 - revoking multiple privileges 448
 - revoking privileges for a list of users 449
 - revoking privileges for a role 449
 - revoking privileges from objects 450
 - revoking privileges from user PUBLIC 451
 - revoking user access 446
 - role privileges 430
 - SELECT privileges 430
 - SQL access privileges 429
 - SQL ROLE 431
 - the PUBLIC user 431
 - unintended effects 441
 - updatable vs non-updatable views 444
 - UPDATE privileges 430
 - using views to restrict data access 443
 - using WITH CHECK OPTION with view privileges 445
- Denial-of-Service attacks 420
- external objects 416
 - external files 416
 - user-defined functions 416
- filesystem
 - Linux/UNIX 415
 - Windows 95/98 and ME 414
 - Windows NT/2000 or XP 414
- firewalls 420
- maintaining the security database 421
- network 419
- owner of the server process on Linux/UNIX 419
- password protection 417
 - isc4.gdb and the masterkey password 417
 - vulnerability of isc4.gdb 418
- restrictions on use of Firebird tools 421



Using Firebird

Alphabetical Index

Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- risks with ISC_USER and ISC_PASSWORD 471
- stored procedures and triggers 549
 - the gsec utility 421
 - error messages 427
 - interactive commands 422
 - ? 426
 - display 423
 - running gsec remotely 422
 - starting a gsec interactive session 422
 - using from a command prompt 426
 - triggers and stored procedures 549
 - Web and other n-tier server applications 419
- security 220
 - access privileges 429–430
 - granting 429–??
 - revoking 446–??
 - roles 438
 - triggers 549
 - REFERENCES privilege 434
 - stored procedures 437–438, 442–443
 - triggers 434
- SELECT 540
 - FOR SELECT vs. 526
 - ORDER BY clause 544
 - views 367
 - WHERE clause 544
- SELECT COUNT (*) 133
- SELECT DISTINCT 141
- select procedures
 - creating 540–545
 - suspending 509
 - terminating 509
- SELECT statements, stored procedures and 506, 526
- SELECT, calling UDFs 585
- Selectable stored procedures 526
- Selectivity of an index 689
- semicolon (;) statement terminator 501
- separators in date and time strings 248
- sequence indicator (triggers) 535
- sequential values 512
- server
 - and database migration steps 618
 - migration, in place 639
 - version compatibility issues 620
- Server context variables 146
 - CURRENT_ROLE 146
 - CURRENT_TIMESTAMP 146
 - CURRENT_USER 146
 - USER 146
- Server Manager 690
- Server name and path 13
- Server programming 494
- Server statistics 452
- servers
 - and databases, migration issues 621
 - migrating 638
- Server-side programming 48
- Service Manager 690
- Service priority on Windows 65
- Services API 47, 690
- Services API, available in all dialects 623
- SET GENERATOR 299
- SET NAMES 310
- SET STATISTICS 360



Using Firebird

Alphabetical Index

Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- SET TERM 189
SET TIME ON, not needed in dialect 3 638
Sets
 glossary entry 690
 output from stored procedures 526
shadowing 379–386
 advantages 380
 automatic 384
 limitations 380
 page size 384
Shadowing and shadows 691
shadows
 conditional 385
 creating 381–385
 defined 379
 dropping 385
 increasing size 386
 modes
 AUTO 384–??
 MANUAL 384–385
 multifile 383
 single-file 382
SHOW command, restrictions 621
SHOW DATABASE 382, 383
SHOW INDEX 354
SHOW TRIGGERS 551
Shutting down a database 471
single quotes
 use for strings 623
 vs double quotes 623, 629
single-column indexes 352
 defined 355
single-file databases 276–277
single-file shadows 382
Singleton SELECTs 506
Single-transaction model 92
SINGULAR operator 149
SMALLINT datatype 236
SMP machines 63
SNAPSHOT 97, 691
SNAPSHOT TABLE STABILITY 97, 691
Soft commit 102
SOME operator 149, 150
sort order, See collation order 302
sorting data 302
specifying
 character sets 257–259, 275
 datatypes 232–233
 domain defaults ??–289
 passwords 275, 279–280
 user names 275, 279–280
specifying collation orders 311–??
SQL
 aliasing columns 141
 comparison operators
 ALL 149
 ANY 149
 BETWEEN 149
 CONTAINING 149
 LIKE 149
 SINGULAR 149
 SOME 150
 STARTING WITH 150
 Data Definition Language (DDL) subset 116



Using Firebird

Alphabetical Index

Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- dialects 114
 - dialect 1 114
 - dialect 2 114
 - dialect 3 114
- DML (Data Manipulation Language) query 119
- dynamic 43
- dynamic SQL (DSQL) 113
- embedded SQL 112
- embedded SQL language 112
- ESQL (embedded SQL) 112
- explicit join syntax 130
- GROUP BY clause 135
 - COLLATE sub-clause 138
 - HAVING sub-clause 137
- implicit join syntax 130
- interactive 114
- internal functions 142
 - AVG() 142
 - CAST() 142
 - COUNT() 142
 - EXTRACT() 142
 - FIRST() SKIP() 142
 - GEN_ID() 143
 - MAX() 143
 - MIN() 143
 - SUBSTRING() 143
 - SUM() 143
 - UPPER() 143
- ISQL (interactive SQL) 114
- JOIN operations 129
 - ambiguity 129
- joins
- conditional 130
- cross join 132
- full join 131
- full outer join 131
- left join 131
- left outer join 131
- natural join 132
- outer 131
- right join 132
- right outer join 132
- logical operators
 - AND 150
 - NOT 150
 - OR 150
- NULL in calculations 126
- operators and expressions 143
- ORDER BY clause 138
- predicates 124
 - elements 124
 - EXISTS() 127, 149
 - IN() 128, 149
 - NULL 125
 - NULL in expressions 125
 - operators 124
 - values 125
- procedural language 112
- PSQL (procedural language) 112
- queries 116
 - queries that call stored procedures 119
 - queries that count rows 133
- SELECT COUNT (*) 133
- SELECT DISTINCT 141



Using Firebird

Alphabetical Index

Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- sets 119
 - cardinality 120
 - cursor sets 122
 - degree 120
 - input 121
 - nested 123
 - output 120
 - output sets as input sets 122
 - sub-selects 123
- SQL-89 implicit join syntax 130
- SQL-92 explicit join syntax 130
- statement list 115
- statement terminators 113
- static vs dynamic 113
- stored procedures and 539
 - specifying variables 502
- table aliasing 129
- the DELETE query 118
- the INSERT query 117
- the SELECT query 117
- the UPDATE query 118
- triggers and 514
- types of statement 111
- UNION query 140
 - UNION ALL 140
- SQL access privileges 430
- SQL clients, specifying character sets 310
- SQL dialects, See dialects 623
- SQL expression elements, listed 144
- SQL functions, listed 142
- SQL Roles 431
- SQL Standard
 - conformance 111
 - SQL-89 111
 - SQL-92 111
 - SQL standard 111
 - SQL warnings, available in all dialects 623
 - SQL> prompt (isql) 159
 - SQL-89 111
 - JOIN syntax 130
 - SQL-92 111
 - JOIN syntax 130
 - SQL-92 compliant features 625
 - SQL-92 standards 42
 - Standard SQL 691
 - statement
 - character set 303
 - statements
 - assignment 505
 - repetitive 507, 526
 - terminating 501
 - Statistics 452
 - cache size 457
 - database dialect 457
 - forced writes setting 457
 - gstat command-line tool 452
 - header page information 456
 - iblockpr and gds_lock_pr 467
 - information about data pages 454
 - lock statistics 467
 - name, location of primary database file 457
 - ODS version 456
 - oldest active transaction 456
 - oldest interesting transaction 456



Using Firebird

Alphabetical Index

Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

on indexes 460
page size 456
record size and version 461
retrieving into programs 464
transaction 108
status, triggers 534
storage requirements, character set 304
Stored procedure 692
Stored procedures 48, 520
 ALTER PROCEDURE 529
 calling from a query 119
 combined selectable/executable 545
 CREATE PROCEDURE 521
 arguments 525
 DROP PROCEDURE 530
 executable 119, 539
 FOR looping 526
 FOR...SELECT...DO...SUSPEND 526
 input parameters 503
 nested 527
 output parameters 503
 RECREATE PROCEDURE 514, 529
 recursive 527
 selectable 119, 526, 540
 using SUSPEND, EXIT, and END 509
 using to view arrays 545
stored procedures 539–548
 altering 529
 arrays and 545–548
 calling 539
 creating 495–501, ??–511
 dependencies

viewing 518
documenting 513
dropping 530
error handling 552–??
 exceptions 549–551, 552
events 513
exiting 509
headers
 output parameters 504
nested 527, 529
overview 520–521
powerful SQL extensions 494
procedure body ??–511
 input parameters 503
 local variables 502, 505
 output parameters 504, 509
 viewing 539
recursive 527, 529
retrieving data 506, 526, 540
return values 504
 incorrect 545
security 437–438, 442–443
suspending execution 509
testing conditions 507, 508
Stored procedures and triggers
 ‘Object is in use’ error 518
 adding comments 513
 assignment statements 505
 begin...end blocks 500
 body elements 496
 case-sensitivity 514
 compiling 515



Using Firebird

Alphabetical Index

Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- CREATEStatement 500
- error trapping and handling 549
- event alerters 512
- flow of control effects 510
- handling exceptions 552
 - Firebird-specific errors 554
 - SQL exceptions 553
 - WHEN...DO 553
- header elements 496
- if...then...else 508
- language elements 496, 497
- managing compiled objects 514
- object dependencies 516
- overview 495
- POST_EVENT 512
- raising an exception 551
- security 549
- singleton SELECTs 506
- statement terminator 501
- statement types not supported 496
- using generators 512
- using SELECT...INTO statements 506
- using variables 502
- while...do 507
- storing
 - Blob IDs 262
- string concatenation operator (||) 147
- String delimiter symbol 17
- string literal
 - character set 303
- Strings
 - concatenation 18
- strings
 - containing quote marks 643
 - delimited with single quotes 623
- structures, database 199, 201
- Subqueries 123
 - correlated 124
- subqueries, comparison operators 148
- Subquery 692
- subscripts (arrays) 267–268
- Sub-select 692
- Sub-selects 123
 - correlated 124
- subtraction operator (-) 148
- SuperServer and Classic architectures, compared 34
- SuperServer architecture 41
- Superserver architecture 692
- Superserver vs Classic 32
- Surrogate key 693
- Surrogate primary key 324
- SUSPEND 509–511
- Sweep Interval 486
- Sweeping 485, 693
 - perform a manual sweep 487
 - sweep interval 486
 - to disable 486
- sweeping databases ??–487
 - sweep interval 488
- switch 394
- switches 399
- Synchronous writes, see Forced writes 476
- syntax
 - assignment statements 505



Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

context variables 504
stored procedures ??–525
System
 requirements 36
System tables 21, 199, 694
system tables 199
system-defined indexes 354, 361
system-defined triggers 538
SystemRestore on Windows ME and XP 476
T
Table 694
table constraints, unnamed 646
Table Reservation 99
Tables 313
 adding a new column to 343
 adding new constraints 344
 ALTER TABLE 340
 altering 340
 choosing indexes 219
 creating 315
 declaring the PRIMARY KEY 325
 declaring the primary key 325
 as a column attribute 325
 as a named table constraint 325
 designing 207
 dropping 349
 dropping an existing column from 345
 dropping columns from 342
 exporting to an external file 340
 EXTERNAL FILE keyword 335
 FOREIGN KEY constraints 326
 importing external files to 337

integrity constraints 211
 Integrity constraints 323
modifying columns in 346
named constraints 333
ownership 315
physical considerations 313
preparing to create 314
PRIMARY KEY constraints 323
RECREATE TABLE 350
removing 349
structural descriptions 313
system 314
UNIQUE keys 326
using external files as 335
tables
 altering 340–349
 caution 343
 circular references 330–331
 creating 315–333
 defined 207
 designing 207
 dropping 349–??
 external 335–340
TCP/IP 72
TCP/IP local loopback connection 77
TCP_NO_NAGLE parameter 65
Temporary directory settings 66
Terminator symbol
 scripts 227
terminators (syntax) 501
testing
 applications 535



Using Firebird

Alphabetical Index

Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

triggers 535
The Firebird Configuration File 62
TIME datatype 244, 247
available only in dialect 3 624
converting from TIMESTAMP 634
older clients cannot access 621
SQL 92 compliant 625
transition feature 619
time indicator (triggers) 535, 537
time string separators 248
TIME, new reserved word 628
TIMESTAMP
display with isql 638
TIMESTAMP datatype 244, 247
available in all dialects 623
converting to DATE or TIME 634
transition feature 619
TIMESTAMP, new reserved word 628
TODAY date literal 249
tokens, unknown 516
TOMORROW date literal 249
top utility (Linux) 9
total_size 588
Transaction 694
Transaction isolation 92, 695
Transaction management 28
Transaction parameter buffer (TPB) 104
Transactions 90
access mode 95
ageing 106
and the API 103
behavior in isql 155

COMMIT 90
COMMIT with RETAIN (COMMIT RETAINING) 102
concurrency 94
context 90
cross-database 93
deadlock 101
embedded 105
isolation 96
isolation level 94
limbo 94
livelock 102
lock resolution 96
locking and lock conflicts 100
management 90
multiple 93
NOWAIT 96
oldest (interesting) 106
oldest active 106
record version 94, 95
recovery of limbo transactions 477
reserving 99
FOR PROTECTED READ 100
FOR PROTECTED WRITE 100
FOR SHARED READ 100
FOR SHARED WRITE 100
ROLLBACK 90
serialization 92
soft commit 102
statistics 106, 108
TPB 104
transaction handles 104
transaction parameter buffer 104



Using Firebird

Alphabetical Index

Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- WAIT 96
- why used 91
- transactions, triggers and 548
- transition features 619
 - DATE, TIME, and TIMESTAMP datatypes 619
 - defined 618
 - double quotes 619
 - large exact numerics 619
- Transitively-dependent 695
- transitively-dependent columns, removing 217
- transliterating character sets 306
- transliteration, defined 308
- Trigger 695
- Triggers 48, 532
 - ALTER TRIGGER 536
 - and transactions 548
 - CREATE TRIGGER 532
 - arguments 533
 - DROP TRIGGER 538
 - NEW and OLD context variables 504
 - system-defined 536
- triggers 532–??
 - access privileges 549
 - altering 536–538
 - creating 495–501, ??–512, 532–??
 - dropping 538
 - duplicating 535
 - firing 534, 549
 - headers 532, 534–535, 537
 - inserting unique values 512
 - isql and 532–??
 - multiple 535
- naming 534
- raising exceptions 550
- referencing values 504
- status 534
- system-defined 538
- testing 535
- transactions and 548
- trigger body ??–512, 532, 535–??, 537
 - context variables 504
- Triggers and stored procedures
 - 'Object is in use' error 518
 - adding comments 513
 - assignment statements 505
 - singleton SELECTs 506
 - begin...end blocks 500
 - body elements 496
 - case-sensitivity 514
 - compiling 515
 - CREATE statement 500
 - error trapping and handling 549
 - event alerters 512
 - flow of control effects 510
 - handling exceptions 552
 - Firebird-specific errors 554
 - SQL exceptions 553
 - WHEN...DO 553
 - header elements 496
 - if...then...else 508
 - language elements 496, 497
 - managing compiled objects 514
 - object dependencies 516
 - overview 495



Using Firebird

Alphabetical Index

Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- POST_EVENT 512
- raising an exception 551
- security 549
- statement terminator 501
- statement types not supported 496
- using generators 512
- using SELECT...INTO statements 506
- using variables 502
- while...do 507
- Troubleshooting
 - BDE 85
 - connection rejected 84
 - connections 82
 - Internet auto-dialup 87
 - IP address vs server name 85
 - network protocol not available 86
 - server permissions 84
- Tuple 695
- TYPE clause of ALTER DOMAIN 294
- TYPE, new reserved word 628
- U**
 - UDF (User-defined function) 696
 - UDF directory
 - configuring 64
 - UDFs
 - allocating memory 576
 - blob 574, 586–590
 - calling 585–586
 - compiling and linking 578
 - creating 574
 - declaring 580–584
 - libraries 573–579
 - modifying libraries 579
 - parameters 574
 - return values 575
 - UDFs (user-defined functions) 572
 - calling a UDF 585
 - with DELETE 586
 - with INSERT 585
 - with SELECT 585
 - with UPDATE 586
 - calling conventions 575
 - compiling and linking a function module 578
 - creating a BLOB control structure 586
 - DECLARE EXTERNAL FUNCTION 574, 580
 - arguments 581
 - declaring a BLOB UDF 588
 - declaring to a database 580
 - declaring with FREE_IT 582
 - developing your own 573
 - existing libraries 572
 - FBUDF 573
 - FreeUDFLib 573
 - ib_udf 573
 - EXTERNAL_FUNCTION_DIRECTORY configuration 583
 - finding declaration details 584
 - library placement 583
 - modifying a library 579
 - security of libraries 583
 - the FBUDF library 593
 - list of functions 593
 - the ib_udf libraries 590
 - list of functions 591



Using Firebird

Alphabetical Index

Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- thread-safe 576
 - writing a BLOB UDF 586
 - Security
 - the gsec utility
 - interactive commands
 - q 426
 - Unbalanced index 696
 - Uninstallation 696
 - Union 697
 - UNION query 140
 - UNION ALL 140
 - UNIQUE constraint 326
 - UNIQUE constraints 208, 211, 323–326, 354
 - unknown tokens 516
 - Updatable view 697
 - updatable views 368–370
 - UPDATE, calling UDFs 586
 - UPDATE, triggers and 504, 532
 - updating
 - data 504
 - views 364, ??–373
 - Use All Space setting 475
 - User name
 - default 11
 - SYSDBA 11
 - user names, specifying 275, 279–280
 - USER variable 146
 - User-defined events 512
 - User-defined functions 28, 49
 - NULL and 126
 - User-defined functions (UDFs) 572
- V**
- validating databases 389, ??–481
 - Validation 697
 - VALUE reserved word 290
 - values
 - assigning to variables 505, 506, 509
 - comparing 148, 504
 - incremental 512
 - referencing 504
 - returned from procedures 504, 545
 - incorrect 545
 - VARCHAR datatype 257, 261, 312
 - variables
 - context 504
 - host-language 506
 - local 503, 505
 - names 513
 - stored procedures 502, 503
 - version
 - client, compatibility issues 620
 - server, compatibility issues 620
 - Version of Firebird server installed 477
 - Versioning architecture 698
 - View 698
 - Views 363
 - columns defined by expressions 368
 - CREATE VIEW 366
 - creating 366
 - dropping 374
 - granting access privileges for 443
 - making read-only views updatable 370
 - modifying a definition 373



Using Firebird

Alphabetical Index

Symbols Numerics A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

naturally updatable 369
privileges 366
read-only 368
security effects of DML operations 445
specifying column names 366
updatable 368
using the SELECTstatement 367
using to restrict access 443
using WITH CHECK OPTION 372
ways to derive 365
why views can be useful 364
views
 advantages ??–364
 altering 373
 column names 367–??
 creating 366–373
 defining columns 144, 368
 dropping 374
 read-only 368–370
 restricting data access 364
 updatable 368–370
 updating 364, ??–373
 with joins 365
virtual tables 366

W

WAIT 96
warnings
 available in all dialects 623
 not available to older clients 624

WEEKDAY, new reserved word 628

WHEN 553, 554

WHEN . . . DO 552

WHILE . . . DO 507

WIN1252 character set 306

Windows applications, character sets 308

Windows clients, specifying character sets 310

Windows Networking (NetBEUI) 74

WITH CHECK OPTION (for views) 372

X

XSQLDA 698
XSQLVAR 699
XSQLVAR field 241

Y

Y valve 699
YEAR, new reserved word 628
YEARDAY, new reserved word 628
YESTERDAY date literal 249