Firebird Database Housekeeping Utility

Norman Dunbar

21 November 2019 – Document version 1.6

Table of Contents

Introduction

Gfix allows attempts to fix corrupted databases, starting and stopping of databases, resolving 'in limbo' transactions between multiple databases, changing the number of page buffers and so on. Gfix is a general purpose tool for system administrators (and database owners) to use to make various 'system level' changes to their databases.

Almost all the gfix commands have the same format when typed on the command line:

gfix [commands and parameters] database_name

The commands and their options are described in the following sections. The database name is the name of the *primary* database file which for a single file database is simply the database name and for multi-file databases, it is the first data file added.

Coming up in the remainder of this manual, we will discuss the following:

- Command line options for the gfix database utility.
- Shadow file handling.
- Cache and buffer handling.
- Transaction management.
- Cache management.
- Starting and stopping a database.
- And much, much more ...

Command Line Options

Running gfix without a command (or an invalid command), or with the new **-?** switch in Firebird 2.5, results in the following screen of helpful information being displayed:

```
usage: gfix [options] <database>
plausible options are:
        -activate       activate shadow file for database usage
        -attach         shutdown new database attachments
        -buffers        set page buffers <n>
        -commit         commit transaction <tr / all>
        -cache          shutdown cache manager
        -full           validate record fragments (-v)
        -force          force database shutdown
        -fetch_password fetch_password from file
        -housekeeping   set sweep interval <n>
        -ignore         ignore checksum errors
        -kill           kill all unavailable shadow files
        -list           show limbo transactions
        -mend           prepare corrupt database for backup
        -mode           read_only or read_write
        -no_update      read-only validation (-v)
        -online         database online <single / multi / normal>
        -prompt         prompt for commit/rollback (-l)
        -password       default password
        -rollback       rollback transaction <tr / all>
```

```
        -sql_dialect    set database dialect n
        -sweep          force garbage collection
        -shut           shutdown <full / single / multi>
        -two_phase      perform automated two-phase recovery
        -tran           shutdown transaction startup
        -use            use full or reserve space for versions
        -user           default user name
        -validate       validate database structure
        -write          write synchronously or asynchronously
        -z              print software version number

    qualifiers show the major option in parenthesis
```

Gfix Commands

> **Note**
>
> In the following discussion, I use the full parameter names in all examples. This is not necessary as each command can be abbreviated. When the command is shown with '[' and ']' in the name then these are the optional characters.
>
> For example, the command **`-validate`** is shown as **`-v[alidate]`** and so can be specified as **`-v`**, **`-va`**, **`-val`** and so on up to the full **`-validate`** version.

For almost all of the options in the following sections, two of the above command line options will be required. These are **`-u[ser]`** and **`-pa[ssword]`**. These can be supplied for every command as parameters on the command line, or can be configured once in a pair of environment variables.

- **`-?`**

  This switch displays the command line options and switches. It replaces the old method in which you had to supply an invalid switch (such as **`-help`**) in order to see the list of valid ones.

  > **Note**
  >
  > Firebird 2.5 onwards.

- **`-FE[TCH_PASSWORD] <password file name> | stdin | /dev/tty`**

  This switch causes the password for the appropriate user to be read from a file as opposed to being specified on the command line. The file name supplied is *not* in quotes and must be readable by the user running gfix. If the file name is specified as `stdin`, then the user will be prompted for a password. On POSIX systems, the file name `/dev/tty` will also result in a prompt for the password.

  > **Note**
  >
  > Firebird 2.5 onwards.

- **`-u[ser] username`**

  Allows the username of the SYSDBA user, or the owner of the database to be specified This need not be supplied if the `ISC_USER` environment variable has been defined and has the correct value.

- **`-pa[ssword] password`**

  Supplies the password for the username specified above. This need not be supplied if the `ISC_PASSWORD` environment variable has been defined and have the correct value.

> **Note**
>
> Up until Firebird 2, any utility which was executed with a password on the command line could result in other users of the server seeing that password using a command like **ps -efx | grep -i pass**. From Firebird 2 onwards, this is no longer the case as the password on the command line can no longer be seen by the **ps** (or other) commands.

To define the username and password as environment variables on a Linux system:

```
linux> export ISC_USER=sysdba
linux> export ISC_PASSWORD=masterkey
```

Alternatively, on Windows:

```
C:\> set ISC_USER=sysdba
C:\> set ISC_PASSWORD=masterkey
```

> **Warning**
>
> This is very insecure as it allows anyone who can access your session the ability to perform DBA functions that you might not want to allow.

- **`-u[ser]`** default user name
- **`-pa[ssword]`** default password

If you have not defined the above environment variables, some commands will not work unless you supply **`-u[ser]`** and **`-pa[ssword]`** on the command line. For example:

```
linux> gfix -validate my_employee
linux> Unable to perform operation. You must be either SYSDBA -
or owner of the database
```

> **Note**
>
> The line that starts with 'Unable to perform' above, has had to be split to fit on the page of the PDF file. In reality, it is a single line.

However, passing the username and password works:

```
linux> gfix -validate my_employee -user sysdba -password masterkey
```

You will notice, hopefully, that some commands do not give any printed output at all. gfix, in the main, only reports when problems are encountered. Always check the response code returned by gfix to be sure that it

worked. However, see the caveats section below for details because it looks like the response code is always zero - at least up until Firebird 2.0.

> **Note**
>
> When logging into a database on a remote server, you will always be required to pass the **-u[ser]** and **-pa[ssword]** parameters.

Shadow Files

A shadow file is an additional copy of the primary database file(s). More than one shadow file may exist for any given database and these may be activated and de-activated at will using the gfix utility.

The following descriptions of activating and de-activating shadow files assume that a shadow file already exists for the database. To this end, a shadow was created as follows:

```
linux> isql my_employee;
SQL> create shadow 1 manual '/home/norman/firebird/shadow/my_employee.shd1';
SQL> create shadow 2 manual '/home/norman/firebird/shadow/my_employee.shd2';
SQL> commit;
SQL> show database;
Database: my_employee
 Owner: SYSDBA
 Shadow 1: "/home/norman/firebird/shadow/my_employee.shd1" manual
 Shadow 2: "/home/norman/firebird/shadow/my_employee.shd2" manual
...
SQL> quit;
```

It can be seen that the database now has two separate shadow files created, but as they are manual, they have not been activated. We can see that shadows are in use if we use gstat as follows:

```
linux> gstat -header my_employee | grep -i shadow
Shadow count 2
```

> **Note**
>
> Sometimes, it takes gstat a while to figure out that there are shadow files for the database.

> **Note**
>
> Shadow file details can be found in the RDB$FILES table within the database.

Activating Shadows

The command to activate a database shadow is:

gfix -ac[tivate] <shadow_file_name>

This makes the shadow file the new database file and the users are able to continue processing data as normal and without loss.

In the event that your main database file(s) become corrupted or unreadable, the DBA can activate a shadow file. Once activated, the file is no longer a shadow file and a new one should be created to replace it. Additionally, the shadow file should be renamed (at the operating system prompt) to the name of the old database file that it replaces.

> **Warning**
>
> It should be noted that activating a shadow while the database itself is active can lead to corruption of the shadow. Make sure that the database file is really unavailable before activating a shadow.

Once a shadow file has been activated, you can see the fact that there are active shadows in the output from gstat:

```
linux> gstat -header my_employee | grep -i shadow
Shadow count 2
Attributes   active shadow, multi-user maintenance
```

> **Note**
>
> The DBA can set up the database to automatically create a new shadow file in the event of a current shadow being activated. This allows a continuous supply of shadow files and prevents the database ever running without one.

Killing Shadows

The command to kill *all unavailable* database shadows, for a specific database, is:

gfix -k[ill] database_name

In the event that a database running with shadow files loses a shadow, or a shadow becomes unusable for some reason, the database will stop accepting new connections until such time as the DBA kills the faulty shadow and, ideally, creates a new shadow to replace the broken one.

The following (contrived) example, shows what happens when the database loses a shadow file and an attempt is made to connect to that database. There are two sessions in the following example, one is connected to the database while the second deletes a shadow file and then tries to connect to the database. The command line prompts shows which of the two sessions we are using at the time.

First, the initial session is connected to the database and can see that there are two shadow files attached:

```
linux_1>isql my_employee
Database: my_employee
SQL> show database;
Database: my_employee
   Owner: SYSDBA
Shadow 1: "/home/norman/firebird/shadow/my_employee.shd1" manual
Shadow 2: "/home/norman/firebird/shadow/my_employee.shd2" manual
 ...
```

In the second session, we delete one of the shadow files, and then try to connect to the database

```
linux_2> rm /home/norman/firebird/shadow/my_employee.shd2
```

```
linux_2> isql_my_employee
Statement failed, SQLCODE = -901
lock conflict on no wait transaction
-I/O error for file "/home/norman/firebird/shadow/my_employee.shd2"
-Error while trying to open file
-No such file or directory
-a file in manual shadow 2 in unavailable
Use CONNECT or CREATE DATABASE to specify a database
SQL> quit;
```

The second session cannot connect to the database until the problem is fixed. The DBA would use the **gfix -k[ill]** command to remove details of the problem shadow file from the database and once completed, the second (and subsequent) sessions would be able to connect.

```
linux_2> gfix -kill my_employee

linux_2> isql my_employee
Database: my_employee
SQL> show database;
Database: my_employee
   Owner: SYSDBA
Shadow 1: "/home/norman/firebird/shadow/my_employee.shd1" manual
...
```

The database now has a single shadow file where before it had two. It is noted, however, that gstat still shows the database as having two shadows, even when one has been removed.

```
linux> gstat -header my_employee | grep -i shadow
Shadow count 2
Attributes   active shadow, multi-user maintenance
```

> **Note**
>
> In addition to the above strange result, if I subsequently DROP SHADOW 1 and COMMIT, to remove the remaining shadow file, gstat now shows that the shadow count has gone up to three when it should have gone down to zero!

Set Database Page Buffers

The database cache is an area of RAM allocated to store (cache) database pages in memory to help improve the efficiency of the database performance. It is far quicker to read data from memory that it is to have to physically read the data from disc.

The size of the database cache is dependent on the database page size and the number of buffers allocated, a buffer is the same size as a database page, and whether the installation is using Classic or Superserver versions of Firebird.

In a Classic Server installation, each connection to the database gets its own relatively small cache of 75 pages while Superserver creates a much larger cache of 2,048 pages which is shared between all the connections.

The command to set the number of cache pages is:

gfix -b[uffers] BUFFERS database_name

This command allows you to change the number of buffers (pages) allocated in RAM to create the database cache.

You cannot change the database page size in this manner, only the number of pages reserved in RAM. One parameter is required which must be numeric and between 50 (the minimum) and 131,072 (the maximum).

The setting applies only to the database you specify. No other databases running on the same server are affected.

The following example shows the use of gstat to read the current number of buffers, the gfix utility being used to set the buffers to 4,000 pages and gstat being used to confirm the setting. The value of zero for page buffers indicates the default setting for the server type is in use.

> **Note**
>
> You can use the gstat command line utility to display the database details with the command line: **gstat -header db_name** however, to run gstat, you need to be logged into the server - it cannot be used remotely.

```
linux> gstat -header my_employee | grep -i "page buffers"
Page buffers    0

linux> gfix -buffers 4000 my_employee

linux> gstat -header my_employee | grep -i "page buffers"
Page buffers 4000
```

Limbo Transaction Management

Limbo transactions can occur when an application is updating two (or more) databases at the same time, in the same transaction. At COMMIT time, Firebird will prepare each database for the COMMIT and then COMMIT each database separately.

In the event of a network outage, for example, it is possible for part of the transaction to have been committed on one database but the data on the other database(s) may not have been committed. Because Firebird cannot tell if these transactions (technically sub-transactions) should be committed or rolled back, they are flagged as being in limbo.

Gfix offers a number of commands to allow the management of these limbo transactions.

> **Note**
>
> The following examples of limbo transactions are based on Firebird 1.5 and have kindly been provided by Paul Vinkenoog. Because of the limitation of my setup, I am unable to create limbo transactions in my current location.
>
> In the spirit of consistency, however, I have renamed Paul's servers and database locations to match the remainder of this document.

Listing Limbo Transactions

The gfix command **`-l[ist]`** will display details of transactions that are in limbo. If there is no output, then there are no transactions in limbo and no further work need be done. The command is:

gfix -l[ist] database_name

An example of listing limbo transactions is shown below. This command is run against the local database on the server named linux where a multi-database transaction had been run connected to databases linux@my_employee and remote:testlimbo. Both of these database names are aliases.

```
linux> gfix -list my_employee
Transaction 67 is in limbo.
 Multidatabase transaction:
 Host Site: linux
 Transaction 67
has been prepared.
 Remote Site: remote
 Database path: /opt/firebird/examples/testlimbo.fdb
```

If the command is run against the remote database then nothing will be listed because that database does not have any limbo transactions - the transaction that went into limbo, when the network failed, for example, was initiated on the local database.

You may also supply the **-p[rompt]** option to the command and you will be prompted to COMMIT or ROLLBACK each detected limbo transaction. In this case, the command would be:

gfix -l[ist] -p[rompt] database_name

An example of this is shown below.

```
linux> gfix -list -prompt my_employee
Transaction 67 is in limbo.
 Multidatabase transaction:
 Host Site: linux
 Transaction 67
has been prepared.
 Remote Site: remote
 Database path: /opt/firebird/examples/testlimbo.fdb
Commit, rollback or neither (c, r, or n)?
```

Committing Or Rolling Back

When a limbo transaction has been detected, the DBA has the option of committing or rolling back one or more of the transactions reported as being in limbo.

When more than one transaction is listed, the DBA can either commit or roll back all transactions in limbo, or a specific transaction number.

The following commands show the **-c[ommit]** option being used, but the **-r[ollback]** option applies as well, it all depends on what the DBA is trying to achieve.

To commit every limbo transaction on the database, the following command would be used:

gfix -commit all database_name

If the DBA wanted to commit a single transaction, then the command would change to the following:

gfix -commit TXN database_name

Where TXN is the transaction number to be committed.

When either of these options are user, there is no feedback from gfix to advise you that the commit actually worked. You would need to rerun the **gfix -list** command to make sure that all, or the selected, limbo transactions had indeed gone.

You cannot commit or rollback a transaction that is not in limbo. If you try , the following will occur:

```
linux> gfix -commit 388 my_employee
failed to reconnect to a transaction in database my_employee
transaction is not in limbo
-transaction 388 is active
unknown ISC error 0
```

When committing or rolling back all limbo transactions, the **-p[rompt]** option can be specified. It is, however, not permitted when processing a single transaction. An example of using the **-p[rompt]** option has been shown above under listing limbo transactions.

Automatic Two-phase Recovery

Gfix can be used to perform automatic two-phase recovery. The command for this is **-t[wo_phase]** and, like **-c[ommit]** and **-r[ollback]** above, requires either 'all' or a transaction number.

The output of the **-l[ist]** command shows what will happen to each listed transaction in the event that the DBA runs the **-t[wo_phase]** command.

The command also takes the **-p[rompt]** option, as above, when used to process all transaction.

The command line to carry out automatic two-phase recovery is:

gfix -t[wo_phase] TXN database_name or

gfix -t[wo_phase] all database_name

As above, TXN is a single transaction number from the list of limbo transactions.

> **Note**
>
> Paul has noted that when using the **-c[ommit]**, **-r[ollback]** or **-t[wo_phase]** options, the output is exactly the same and appears to show that these three are all just synonyms for the **-l[ist] -p[rompt]** pair of options. This occurred whether or not Paul used the transaction number, 67, or 'all' in the command line.

Cache Manager

When the help page for gfix is displayed there is a message in the output for the **-ca[che]** option which states:

```
...
-ca[che]      shutdown cache manager
...
```

However, when called this option simply displays the help page again.

The question that immediately springs to my mind is, if we can shutdown the cache manager with this option, how do we start it back up again?

Changing The Database Mode

Databases can be set to run in one of two modes, read only - where no updates are permitted, and read/write - where both reading and writing of data is permitted. By default, Firebird creates read/write databases and as such, all read/write databases must be placed on a file system which allows writing to take place.

Should you wish to put a Firebird database on a CD, for example, you wouldn't be able to do so. After a new database has been populated with data it can be changed to read only mode, and then used on a CD (or other read only file systems) with no problems.

> **Note**
>
> Firebird uses SQL internally to maintain its internal structures with details about transactions, for example, and this is the reason that a database must be placed on a read/write file system regardless of whether only SELECT statements are run or not.

> **Note**
>
> Only databases in dialect 3 can be changed to read only mode.

The command to set the required mode for a database is:

gfix -mo[de] MODE database_name

The command takes two parameters, the MODE which must be one of the following:

- **`read_only`** - the database cannot be written to.
- **`read_write`** - the database can be written to.

The meaning of the two modes should be quite meaningful.

The second parameter is a database name to apply the mode change to.

The following example shows how to put a database into read only mode, and then change it back again. The example also shows what happens when you try to update the database while running in read only mode.

```
linux> gfix -mode read_only my_employee

linux> isql my_employee
Database: my_employee

SQL> create table test(stuff integer);
Statement failed, SQLCODE = -902
Dynamic SQL Error
-attempted update on read-only database
```

```
SQL> quit;

linux> gfix -mode read_write my_employee

linux> isql my_employee
Database: my_employee

SQL> create table test(stuff integer);

SQL> show table test;
STUFF      INTEGER Nullable

SQL> quit;
```

If there are any connections to the database in read/write mode when you attempt to convert the database to read only, the attempt will fail as shown below with Firebird 1.5.

```
linux> gfix -mode read_only my_employee
lock time-out on wait transaction
-lock time-out on wait transaction
-object my_employee is in use

linux> echo $?
0
```

> **Warning**
>
> As with many failures of gfix, the response code returned to the operating system is zero.

Under Firebird 2, the error message is more self explanatory:

```
linux> gfix -mode read_only my_employee
lock time-out on wait transaction
-object /opt/firebird/databases/my_employee.fdb is in use

linux> echo $?
0
```

Setting The Database Dialect

The dialect of the database is simply a term that defines the specific features of the SQL language that are available when accessing that database. There are three dialects at present (Firebird version 2.0), these are:

- Dialect 1 stores date and time information in a DATE data type and has a TIMESTAMP data type which is identical to DATE. Double quotes are used to delimit string data. The precision for NUMERIC and DECIMAL data types is less than a dialect 3 database and if the precision is greater than 9, Firebird stores these as DOUBLE PRECISION. INT64 is not permitted as a data type.
- Dialect 2 is available only on the Firebird client connection and cannot be set in the database. It is intended to assist debugging of possible problems with legacy data when migrating a database from dialect 1 to 3. This dialect cannot be set for a database using gfix. (See below.)
- Dialect 3 databases allow numbers (DECIMAL and NUMERIC data types) to be stored as INT64 when the precision is greater than 9. The TIME data type is able to be used and stores time data only. The DATE data

type stores on date information. Double quotes can be used but only for identifiers that are case dependent, not for string data which has to use single quotes.

The command to change the SQL dialect for a database is:

gfix -s[ql_dialect] DIALECT database_name

The DIALECT parameter is simply 1 or 3.

The following example changes a database to use dialect 3 which will allow many newer features of SQL 92 to be used.

```
linux> gfix -sql_dialect 3 my_employee

linux> gstat -header my_employee | grep dialect
Database dialect    3

linux> gfix -sql_dialect 1 my_employee

linux> gstat -header my_employee | grep dialect
Database dialect    1
```

Because you cannot use gstat remotely, you may also use the isql command SHOW SQL DIALECT from a remote location to see which dialect your client and database are using, as follows:

```
remote> isql my_employee -user norman -password whatever
Database: my_employee

SQL> show sql dialect;
Client SQL dialect is set to: 3 and database SQL dialect is: 3
```

Although dialect 2 is possible on the client, trying to set a dialect of 2 will fail on the server as the following example shows.

```
linux> gfix -sql_dialect 2 my_employee
Database dialect 2 is not a valid dialect.
-Valid database dialects are 1 and 3.
-Database dialect not changed.
```

To set dialect 2 for your *client* connection, you use isql as follows:

```
linux> isql my_employee
Database: my_employee

SQL> set sql dialect 2;
WARNING: Client SQL dialect has been set to 2 when connecting -
to Database SQL dialect 3 database.

SQL> show sql dialect;
Client SQL dialect is set to: 2 and database SQL dialect is: 3
```

> **Note**
>
> The WARNING line above has had to be split to fit on the page of the PDF version of this manual. In reality, it is a single line of text.

Database Housekeeping And Garbage Collection

Garbage

Garbage, for want of a better name, is the detritus that Firebird leaves around in the database after a rollback has been carried out. This is basically a copy of the row(s) from the table(s) that were being updated (or deleted) by the transaction prior to the rollback.

Almost all garbage is created by committed transactions. Since around V2.5 transactions that rollback are cleaned up immediately - assuming that Firebird is still running.

The major cause of garbage build-up is long running transactions that require Firebird to keep old versions of records that are frequently updated. Another source of garbage is an application strategy that deletes records and never revisits them.

What actually happens on delete is that Firebird stores a "deleted stub" with the full record as a back version. Until the delete is mature - meaning that all active transactions started *after* the delete was committed - the old version must be preserved.

Imagine a table that's indexed and accessed by date. On some schedule, records age out and are deleted. In the application, records are accessed by date and the deleted records are so old no query every asks for them. So there they sit, taking up space and doing no good to anyone. Even with a garbage collect thread, some active transaction has to *stumble* over deleted records or records with unneeded back versions before the record will be garbage collected.

In cooperative garbage collection, that particular record will be cleaned up immediately (or at least when the transaction gets some cycles). The dedicated garbage collection thread should clean up all the records on a page, but not until an active transaction tells it that there's a page that needs cleaning.

Because Firebird uses multi-generational architecture, every time a row is updated or deleted, Firebird keeps a copy in the database. These copies use space in the pages and can remain in the database for some time, especially if there are no active transactions stumbling across them!

There are a number of causes of garbage:

- Remnants from a committed transaction. This is the main cause of garbage since around Firebird version 2.5.
- Remnants from an aborted (rolled back) transaction *may* exist in Firebird versions prior to 2.5, newer versions perform immediate clean up after a rollback however, if the Firebird Server, the Operating System or the physical server crashed, then these remnants may still exists, even in later versions of Firebird.
- Applications, described above, which delete database records, but then, subsequently, never revisit those deleted versions to garbage collect them automatically.

With regard to the remnants from aborted or rolled back transactions, Firebird (now) carries out record keeping to facilitate save points. This housekeeping allows Firebird to identify and, if necessary, undo all changes made by a transaction in the event that it is rolled back, or which failed due to a lost connection.

If a failed transaction is rolled back in either manner, its state is set to *committed* as there are no differences between a failed transaction and one in which it committed after making no changes.

These remnants are simply older copies of the rows that were being updated by the respective transactions. The differences are that:

- Whenever a subsequent transaction reaches garbage from a *committed* transaction, that garbage is automatically cleared out, but see above for reasons where this may not take place often enough.
- Rolled back garbage looks just like record versions created by active transactions. Those records can be accessed either sequentially (during a full table scan) or by index - assuming that the index entry was made before the crash that left the garbage around. The index entries will exist in the case of all but the last change made. When one transaction reads a record version created by a transaction that's listed in the transaction bit vector as active, the reader attempts to get a lock on the apparently active transaction id. If the lock request succeeds, then the other transaction is dead and the reader will either clean up the mess or notify the garbage collect thread to do so.

Firebird will automatically sweep through the database and remove the remnants of rolled back transactions and this has two effects:

- The space recovered is made available for reuse by the same table, however, if this results in the page becoming completely empty, then it can be used for any purpose within the database.
- The performance of the database may be affected while the sweep is in progress.

> **Note**
>
> One other method of clearing out old rolled back transactions' garbage is simply to carry out a database backup. Gbak reads every table sequentially and thus visits every row in every table. Applications which also visit every row in one or more tables, will also cause the garbage in those tables to be collected.
>
> Neither will affect the database's OIT (Oldest Intersting Transaction) or OST (Oldest Snapshot) settings however.

In the Super Server version of Firebird 2.0, garbage collection has been vastly improved. There are now three different ways of operation and these are configurable by setting the *GCPOLICY* parameter in the `firebird.conf` configuration file. By default, Super Server uses *combined* while Classic Server uses *cooperative*. The other option is *background*.

> **Note**
>
> Classic Server ignores the setting and always uses cooperative garbage collection.

Record Versions

Normally, when a "back" or old version of a row in a table is created, it will be stored on the same page as the newset version. This is usually fine as the back version is not normally a complete copy of the old version, merely a list of differences from the newest version. Enough information is retained in the old version, to be able to recreate it, if necessary.

If the database is suffereing from a lack of garbage colecting, either deliberately, or down to the application design, then it is possible that there will be a build up of enough back versions to fill the target page. When the chain of old versions gets too big, Firebird has to move the old versions to a diffreent page which, if it occurs as part of an UPDATE statement, as it normally will, the UPDATE will run a lot slower than usual and will greatly increase the cost of subsequent garbage collection againts that table.

Cooperative Garbage Collection

This is the default setting, indeed the only setting, that Classic Server uses. In this mode, the normal operation - as described above - takes place. When a full scan is performed (perhaps during a backup) old versions of the rows are deleted at that point in time. Record versions which are old enough that no active transactions have any interest in them will be removed, as will any versions created by failed transactions, if there are any present. (Which there shouldn't be!)

Background garbage Collection

Super Server has, even since before version 1.0, performed background garbage collection where the server informs the garbage collector about old versions of updated and deleted rows when they are ready to be cleaned up. This helps avoid the need to force a full scan of each record in the database tables to get the garbage collector to remove these old versions. An active transaction has to recognize the need for garbage collection and notify the server which puts that record id on a list for the garbage collect thread. So an unvisited record will not attract the garbage collector unless another record on that page is read and does need cleanup.

When all rows in a table are read by the server, any old record versions are flagged to the garbage collector as being ready to be cleared out. They are not deleted by the scanning process as in the cooperative method. The garbage collector runs as a separate background thread and it will, at some point, remove these older record versions from the database.

Combined Garbage Collection

This is the default garbage collection method for Super Server installations. In this mode, both the above methods are used together.

Setting Sweep Interval

The default sweep interval for a new database is 20,000. The sweep interval is the *difference* between the *Oldest Snapshot Transaction*, or OST and the *Oldest Interesting Transaction* or OIT.

> **Note**
>
> This doesn't mean that every 20,000 transaction a sweep will take place. It will take place when the *difference* between the OST and the OIT is greater than the sweep interval.

An interesting transaction is one which has not yet committed. It may be still active, in limbo or may have been rolled back. (Limbo transactions are never garbage collected.)

The sweep facility runs through the database and gets rid of old rows in tables that are out of date. This prevents the database from growing too big and helps reduce the time it takes to start a new transaction on the database.

> **Note**
>
> If you find that starting a new transaction takes a long time, it may be a good idea to run a manual sweep of the database in case the need for a sweep is causing the hold-up.

You can check if a manual sweep may be required by running the gstat utility to check the database header page and extract the Oldest Transaction (OIT) and Oldest Snapshot (OST) numbers from the output. If OST - OIT is small (less than the sweep interval) then a manual sweep may be in order. The SHOW DATABASE command in isql will also show the details you need.

Alternatively, another idea is to run gstat with the switches set to show old record versions. If that shows a problem, then it may be a good idea to start looking for long running transactions.

The options for this are:

- **gstat <database> -r[ecord]**
- **gstat <database> -d[ata] -r[ecord]**
- **gstat <database> -r[ecord] -t[able] <table_names>**

For example:

```
tux> gstat test.fdb -r -t NORMAN

...
Analyzing database pages ...
NORMAN (142)
    Primary pointer page: 268, Index root page: 269
    Average record length: 0.00, total records: 15
    Average version length: 9.00, total versions: 15, max versions: 1
    Data pages: 1, data page slots: 1, average fill: 16%
...
```

The information is shown in the 'record versions' statistic. In this example, there are 15 versions and as the 'total records' is also 15, then all the records have been deleted and need gargabge collecting.

A manual sweep can be run by using the **`-s[weep]`** command. (See below).

To alter the database's automatic sweep interval, use the following command:

gfix -h[ousekeeping] INTERVAL database_name

The INTERVAL parameter is the new value for the sweep interval. The database name parameter is the database upon which you wish to alter the setting for automatic sweeping. The following example shows the setting being changed from the default to a new value of 1,000.

```
linux> gfix -h 1000 my_employee

linux> gstat -header my_employee | grep Sweep
Sweep interval:     1000
```

Manual Garbage Collection

If automatic sweeping has been turned off, or only runs rarely because of the sweep interval setting, the DBA can manually force a sweep to be performed. The command to carry out this task is:

gfix -s[weep] [-i[gnore]] database_name

This command will force the garbage left over from old rolled back transactions to be removed, reducing the database size and improving the performance of new transactions. .Rolled back transactions are less of a problem

than old versions from committed transactions, however, when the newest versions is being used by all current and future active transactions.

The **-i[gnore]** option may be supplied. This forces Firebird to ignore checksum errors on database pages. This is not a good idea and should rarely need to be used, however, if your database has suffered some problems it might be necessary to use it.

> **Note**
>
> Checksums have not been used for a number of years as it was found that a significant percentage of CPU was consumed by check summing to find partial page writes - none of which were ever found!

The following example shows a manual database sweep being implemented:

```
linux> gfix -sweep my_employee
```

Disabling Automatic Sweeping

If you set the sweep interval to zero then automatic sweeping will be disabled. This implies that there will be no automatic housekeeping done so your database performance will not suffer as a result of the processing requirements of the automatic sweep.

If you disable sweeping you are advised to run a manual sweep at regular intervals when the database is quiet. Alternatively, simply make sure that you take regular backups of the database and as this is something you should be doing anyway, it shouldn't be a problem.

Database Startup and Shutdown

> **Note**
>
> The first part of this section describes the shutdown and startup options up to Firebird 2.0. There is a separate section at the end which discusses the new *states* for starting and stopping a database using Firebird 2.0 onwards.

Database Shutdown

If there is maintenance work required on a database, you may wish to close down that database under certain circumstances. This is different from stopping the Firebird server as the server may well be running other databases which you do not wish to affect.

The command to close a database is:

gfix -shut OPTION TIMEOUT database_name

The TIMEOUT parameter is the time, in seconds, that the shutdown must complete in. If the command cannot complete in the specified time, the shutdown is aborted. There are various reasons why the shutdown may not complete in the given time and these vary with the mode of the shutdown and are described below.

The OPTION parameter is one of the following:

- **-at[tach]** - prevents new connections.
- **-tr[an]** - prevents new transactions.
- **-f[orce]** - simply aborts all connections and transactions.

When a database is closed, the SYSDBA or the database owner can still connect to perform maintenance operations or even query and update the database tables.

> **Note**
>
> If you specify a long time for the shutdown command to complete in, you can abort the shutdown by using the **-online** command (see below) if the timeout period has not completed.

Preventing New Connections

-at[tach] : this parameter prevents any new connections to the database from being made with the exception of the SYSDBA and the database owner. The shutdown will fail if there are any sessions connected after the timeout period has expired. It makes no difference if those connected sessions belong to the SYSDBA, the database owner or any other user. Any connections remaining will terminate the shutdown with the following details:

```
linux> gfix -shut -attach 5 my_employee
lock conflick on no wait transaction
-database shutdown unsuccessful
```

Anyone other than the SYSDBA or database owner, attempting to connect to the database will see the following:

```
linux> isql my_employee -user norman -password whatever
Statement failed, SQLCODE = -901
database my_employee shutdown
Use CONNECT or CREATE DATABASE to specify a database
SQL>
```

Connections in the database will still be able to start new transactions or complete old ones.

Preventing New Transactions

-tr[an] : prevents any new transactions from being started and also prevents new connections to the database. If there are any active transactions after the timeout period has expired, then the shutdown will fail as follows:

```
linux> gfix -shut -tran 5 my_employee
lock conflick on no wait transaction
-database shutdown unsuccessful
```

If any user connected to the database being shutdown with the **-tr[an]** tries to start a new transaction during the shutdown timeout period, the following will result:

```
SQL> select * from test;
Statement failed, SQLCODE = -902
database /home/norman/firebird/my_employee.fdb shutdown in progress
Statement failed, SQLCODE = -902
```

```
database /home/norman/firebird/my_employee.fdb shutdown in progress
Statement failed, SQLCODE = -901
Dynamic SQL Error
-SQL error code = -901
-invalid transaction handle (expecting explicit transaction start)
```

Force Closure

`-f[orce]` : shuts down with no regard for the connection or transaction status of the database. No new connections or transactions are permitted and any active sessions are terminated along with any active transactions.

Anyone other than SYSDBA or the database owner trying to connect to the database during the timeout period will not be able to connect successfully or start any (new) transactions.

Be nice to your users, use the **`-f[orce]`** option with great care.

> **Warning**
>
> There is a bug in Classic Server which still exists at version 2.0. The bug is such that the **`-f[orce]`** option behaves in exactly the same way as the **`-at[tach]`** option.

Starting a Database

Once all maintenance work required on a database has been carried out, you need to restart the database to allow normal use again. (See shutdown option above for details of closing a database.)

The **`-o[nline]`** command allows a database to be restarted. It takes a single parameter which is the database name as follows:

gfix -o[nline] database_name

The following example shows a closed database being started.

```
linux> gfix -online my_employee
```

New Startup and Shutdown States in Firebird 2.0

The above discussion of stopping and starting a database apply to all versions of the server up to version 2.0. From 2.0 the commands will work as described above, but a new *state* has been added to define exactly how the database is to be stopped or started. The commands change from those described above to the following:

gfix -shut STATE OPTION TIMEOUT database_name

gfix -o[nline] STATE database_name

STATE is new in Firebird 2.0 and is one of the following:

- **`normal`** - This is the default state for starting the database backup. It allows connections from any authorised users - not just SYSDBA or the database owner. This option is not accepted for shutdown operations.

- **`multi`** - this is the default mode as described above. When the database is shutdown as above, or using the multi state, then *unlimited* connections can be made by the SYSDBA or the database owner. No other connections are allowed.
- **`single`** - Similar to the multi option above, but only *one* SYSDBA or database owner connection is allowed.
- **`full`** - Shutdown and don't allow *any* connections from anyone, even SYSDBA or the database owner. This is not an acceptable option for starting up a database.

> **Note**
>
> There is no leading dash for the state parameters, unlike the command itself and the **`-shut`** OPTION.

There is a hierarchy of states for a database. The above list shows them in order with normal at the top and full at the bottom.

This hierarchy is important, you cannot *shutdown* a database to a *higher or equal* level that it currently is, nor can you *startup* a database to a *lower or equal* level.

If you need to identify which level a database is currently running at, gstat will supply the answers. The following example puts a database fully online then progressively shuts it down to fully offline. At each stage, gstat is run to extract the Attributes of the database.

```
linux> gfix -online normal my_employee
linux> gstat -header my_employee | grep Attributes

        Attributes

linux> gfix -shut multi -attach 0 my_employee
linux> gstat -header my_employee | grep Attributes

        Attributes              multi-user maintenance

linux> gfix -shut single -attach 0 my_employee
linux> gstat -header my_employee | grep Attributes

        Attributes              single-user maintenance

linux> gfix -shut full -attach 0 my_employee
linux> gstat -header my_employee | grep Attributes

        Attributes              full shutdown

linux>
```

Database Page Space Utilisation

When a database page is being written to, Firebird reserves 20% of the page for future use. This could be used to extend VARCHAR columns that started off small and then were updated to a longer value, for example.

If you wish to use all the available space in each database page, you may use the **`-use`** command to configure the database to do so. If you subsequently wish to return to the default behaviour, the **`-use`** command can be used to revert back to leaving 20% free space per page.

> **Note**
>
> Once a page has been filled to 'capacity' (80 or 100%) changing the page usage setting will not change those pages, only subsequently written pages will be affected.

The **-use** command takes two parameters as follows:

gfix -use USAGE database_name

The USAGE is one of:

- **full** : use 100% of the space in each database page.
- **reserve** : revert to the default behaviour and only use 80% of each page.

The following example configures a database to use all available space in each database page:

```
linux> gfix -use full my_employee
linux> gstat -header my_employee | grep Attributes
Attributes no reserve
```

The following example sets the page usage back to the default:

```
linux> gfix -use reserve my_employee
linux> gstat -header my_employee | grep Attributes
Attributes
```

If you are using full page utilisation then the Attributes show up with 'no reserve' in the text. This doesn't appear for normal 80% utilisation mode.

Database Validation and Recovery

Database Validation

Sometimes, databases get corrupted. Under certain circumstances, you are advised to validate the database to check for corruption. The times you would check are:

- When an application receives a *database corrupt* error message.
- When a backup fails to complete without errors.
- If an application aborts rather than shutting down cleanly.
- On demand - when the SYSDBA decides to check the database.

> **Note**
>
> Database validation requires that you have exclusive access to the database. To prevent other users from accessing the database while you validate it, use the **gfix -shut** command to shutdown the database.

When a database is validated the following checks are made *and corrected* by default:

- Orphan pages are returned to free space. This updates the database.
- Pages that have been misallocated are reported.
- Corrupt data structures are reported.

There are options to perform further, more intensive, validation and these are discussed below.

Default Validation

The command to carry out default database validation is:

gfix -v[alidate] database_name

This command validates the database and makes updates to it when any orphan pages are found. An orphan page is one which was allocated for use by a transaction that subsequently failed, for example, when the application aborted. In this case, committed data is safe but uncommitted data will have been rolled back. The page appears to have been allocated for use, but is unused.

This option updates the database and fixes any corrupted structures.

Full Validation

By default, validation works at page level. If no need to go deeper and validate at the record level as well, the command to do this is:

gfix -v[alidate] -full database_name

using this option will validate, report and update at both page and record level. Any corrupted structures etc will be fixed.

Read-only Validation

As explained above, a validation of a database will actually validate and update the database structures to, hopefully, return the database to a working state. However, you may not want this to happen and in this case, you would perform a read only validation which simply reports any problem areas and does not make any changes to the database.

To carry out a read only validation, simply supply the **`-n[o_update]`** option to whichever command line you are using for the validation. To perform a full validation, at record and page level, but in reporting mode only, use the following command:

gfix -v[alidate] -full -n[o_update] database_name

On the other hand, to stay at page level validation only, the command would be:

gfix -v[alidate] -n[o_update] database_name

Ignore Checksum Errors

Checksums are used to ensure that data in a page is valid. If the checksum no longer matches up, then it is possible that a database corruption has occurred. You can run a validation against a database, but ignore the checksums using the **`-i[gnore]`** option.

This option can be combined with the **`-n[o_update]`** option described above and applies to both full and default validations. So, to perform a full validation and ignore checksums on a database, but reporting errors only, use the following command:

gfix -v[alidate] -full -i[gnore] -n[o_update] database_name

Alternatively, to carry out a page level validation, ignoring checksum errors but updating the database structures to repair it, the command would be:

gfix -v[alidate] -i[gnore] database_name

Ignoring checksums would allow a corrupted database to be validated (unless you specify the **-n[o_update]** option) but it is unlikely that the recovered data would be usable, if at all, present.

Database Recovery

If the database validation described above produces no output then the database *structures* can be assumed to be valid. However, in the event that errors are reported, you may have to repair the database before it can be used again.

Recover a Corrupt Database

The option required to fix a corrupted database is the **gfix -m[end]** command. However, it cannot fix all problems and *may result in a loss of data*. It all depends on the level of corruption detected. The command is:

gfix -m[end] database_name

This causes the corruptions in data records to be ignored. While this sounds like a good thing, it is not. Subsequent database actions (such as taking a backup) will not include the corrupted records, leading to data loss.

> **Important**
>
> The best way to avoid data loss is to make sure that you have enough regular backups of your database and to regularly carry out test restorations. There is no point taking backups every night, for example, if they cannot be used when required. Test always and frequently.
>
> Equally, when attempting to recover a potentially corrupted database, *always* work with a copy of the main database file and never with the original. Using the **-mend** option can lead to silent deletions of data because gfix doesn't care about internal database constraints like foreign keys etc, the **-mend** option simply says to gfix "*go ahead and clean out anything you don't like*".

Database Write Mode

Many operating systems employ a disc cache mechanism. This uses an area of memory (which may be part of your server's overall RAM or may be built into the disc hardware) to buffer writes to the hardware. This improves the performance of applications that are write intensive but means that the user is never certain when their data has actually been written to the physical disc.

With a database application, it is highly desirable to have the data secured as soon as possible. Using Firebird, it is possible to specify whether the data should be physically written to disc on a COMMIT or simply left to the operating system to write the data *when it gets around to it*.

To give the DBA or database owner full control of when data is written, the **gfix -w[rite]** command can be used. The command takes two parameters:

gfix -write MODE database_name

The MODE parameter specifies whether data would be written immediately or later, and is one of:

- **sync** - data is written synchronously. This means that data is flushed to disc on COMMIT. This is safest for your data.
- **async** - data is written asynchronously. The operating system controls when the data is actually written to disc.

If your system is highly robust, and protected by a reliable UPS (uninterruptable Power Supply) then it is possible to run asynchronously but for most systems, synchronous running is safest this will help prevent corruption in the event of a power outage or other uncontrolled shutdown of the server and/or database.

> **Note**
>
> Firebird defaults to synchronous mode (forced writes enabled) on Linux, Windows NT, XP, 2000, 2003 and Vista.

This command has no effect on Windows 95, 98 and ME.

> **Warning**
>
> Cache flushing on Windows servers (up to but not including Vista - which has not been confirmed yet) is unreliable. If you set the database to **async** mode (forced writes disabled) then it is possible that the cache will never be flushed and data could be lost if the server is never shutdown tidily.

> **Warning**
>
> If your database was originally created with Interbase 6 or an early beta version of Firebird then the database will be running in asynchronous mode - which is not ideal.

Version Number

The **-z** option to gfix simply prints out the version of the Firebird utility software that you are running. It takes no parameters as the following example (running on Linux) shows.

```
linux> gfix -z
gfix version LI-V2.0.0.12748 Firebird 2.0
```

Caveats

This section summarises the various problems that you may encounter from time to time when using gfix. They have already been discussed above, or mentioned in passing, but are explained in more details here.

Shadows

The gstat seems to take some time to respond to the addition of shadow files to a database. After adding two shadows to a test database, gstat still showed that there was a Shadow count of zero.

Even worse, after killing the second shadow file and running the DROP SHADOW command in isql to remove the one remaining shadow file, gstat decided that there were now three shadow files in use.

Response Codes Are Usually Zero

Even using Firebird version 2 it appears that many commands, which fail to complete without an error, return a response of 0 to the operating system.

> **Note**
>
> This problem was fixed in Firebird 2.1 RC1. It has been tested and a successful operation returns zero to the shell while a failure returns 1.
>
> This section will remain in the manual as there are still a large number of users with older versions of Firebird.

For example, the following shows two attempts to shut down the same database, the second one should fail - it displays an error message - but still returns a zero response to the operating system. This makes it impossible to built correctly error trapped database shutdown scripts as you can never tell whether it actually worked or not.

```
linux> gfix -shut -force 5 my_employee
linux> echo $?
0

linux> gfix -shut -force 5 my_employee
Target shutdown mode is invalid for database -
"/home/norman/firebird/my_employee.fdb"
linux> echo $?
0
```

> **Note**
>
> As mentioned above, this is no longer a problem from release 2.1 RC1 onwards. The second attempt to close the database will correctly return 1 to the shell.

Force Closing a Database

Under classic server, using the **`-f[orce]`** option to the **`-shut`** command acts exactly the same as the **`-at[tach]`** option.

Limbo Transactions

There are a couple of problems with limbo transactions as discovered by Paul in his testing.

Limbo Transaction Options - All The Same?

When processing limbo transactions, it appears under Firebird 1.5 at least, that the **-l[ist] -p[rompt]** option is called regardless of whether you use **-c[ommit]**, **-r[ollback]** or **-t[wo_phase]**. The outcome is the same regardless of whether the DBA specifies a specific transaction number or 'all' on the command line - a prompt is given with the option to commit, rollback or neither.

Limbo Transactions - Can Be Backed Up

Paul's testing of limbo transactions revealed that it is possible to make a backup of a database with limbo transactions. This backup can then be used to create a new database and the limbo transactions will still be able to be listed. This applies to a file system copy of the database and to version 1.5 of Firebird.

If you attempt to list the limbo transactions in the copy database *and* the original database has been deleted, renamed or has been set to read-only, then gfix will present you with a request to supply the correct path to the original database

```
linux>cd /home/norman/firebird
linux>cp my_employee.fdb my_new_employee.fdb

linux> mv my_employee.fdb my_old_employee.fdb

linux> gfix -list /home/norman/firebird/my_new_employee.fdb
Transaction 67 is in limbo.
Could not reattach to database for transaction 67.
Original path: /home/norman/firebird/my_employee.fdb

Enter a valid path: /home/norman/firebird/my_old_employee.fdb

 Multidatabase transaction:
 Host Site: linux
 Transaction 67
has been prepared.
 Remote Site: remote
 Database path: /opt/firebird/examples/testlimbo.fdb
```

In the above example, the original database my_employee.fdb was first of all copied using the operating system command **cp** to my_new_employee.fdb and then renamed to my_old_employee.fdb.

Gfix was then run on the copy named my_new_employee.fdb and it noted the limbo transaction. However, it could not find the original database file as it had been renamed, so gfix prompted for the path to the original database file. When this was entered, gfix happily listed the details.

> **Warning**
>
> This implies that if you have a database with limbo transactions and you copy it using the operating system utilities and subsequently run gfix against the new database, it is possible to have gfix fix limbo transactions in the original database file and not in the one you think it is updating - the copy.
>
> It is also a good warning about making copies of databases without using the correct tools for the job.

Appendix A: Document history

The exact file history is recorded in the `manual` module in our CVS tree; see http://sourceforge.net/cvs/?group_id=9028. The full URL of the CVS log for this file can be found at http://firebird.cvs.sourceforge.net/viewvc/firebird/manual/src/docs/firebirddocs/fbutil_gfix.xml?view=log

Revision History

1.0	19 June 2007	ND	Created as a chapter in the Command Line Utilities manual.
1.1	20 October 2009	ND	More minor updates and converted to a stand alone manual.
1.2	25 June 2010	ND	Fixed spacing on a couple of lists. Added an enhancement to the details of the -mend recovery option. It can lead to a loss of data.
1.3	11 October 2011	ND	Spelling errors corrected. Updated for Firebird 2.5.
1.4	09 April 2013	ND	Updated to note that gfix returns correct error codes to the shell from release 2.1 RC1 onwards.
1.5	13 February 2018	ND	DOC-129 - Updated to correct details of the Sweep Interval and how to check what the current interval is.
1.6	21 November 2019	ND	Updated the Garbage section to better explain garbage causes etc. Courtesy of Ann Harrison.

Appendix B: License notice