this work was produced by Paul Beach and IBPhoenix Inc.

Diagnosing and Repairing InterBase Database Corruption

Updated and revised 29th Sep 2000 by Paul Beach

A number of types and kinds of database corruption can be repaired with gfix (alice) and gbak (burp). However, it is possible in some rare cases that a database file may be corrupted beyond the ability of these tools to repair the damage. In such an instance more drastic measures may be needed to get the database back on line. If you try and repair your database and it fails, please contact us and we will see what we can do to help.

The most frequent cause of corruption is either an abrupt or catastrophic power loss on the database server. Shutting off the power when an application is in the process of writing to the database can result in corrupted or incomplete data being written to the database file. In all cases the database user, and database administrator should take every possible precaution to prevent this happening.

The InterBase Server has two write modes (forced writes), synchronous and asynchronous. Pre InterBase V6.0 the default write mode was synchronous:

```
gfix -write sync database.gdb
```

Post InterBase V6.0 the write mode is asynchronous:

```
gfix -write async database.gdb
```

Synchronous writes are known as "careful writes", in that the InterBase engine will flush modified pages to disk on a transaction commit, and will write the pages back to the database in the correct order (as far as the database server is concerned) and so minimise any possible data loss. Careful write is present in all cases, though without forced write, it's careful only up to the Operating System file cache. Forced write has no effect on Windows 3.1, Windows 95 and Windows 98. On Unix and NT, it causes the operating system to bypass its file cache and send the page directly to disk.

The normal mode on Unix systems has forced write off because it is a significant performance cost. The normal mode on NT is forced writes on because the Operating System is too flaky to trust with valuable pages. Prior to InterBase V6, all I/O was synchronous, meaning that when a page was read or written, the thread waited until the operating system said that the operation was done. That's a reasonable way to manage careful writes, in most cases, and it appears to be the way writes are done, in all cases. Asynchronous reads are new in InterBase V6 and allow the server to continue processing on behalf of a thread while the read is in progress. Unix systems don't support asynchronous I/O generally, so this is an NT only optimization. Careful writes are the way the world is. Forced writes can be turned off and on by the user. The server makes its own choices between synchronous and asynchronous I/O, never trading reliability for speed.

Typically most users turn forced writes off, for the performance gains that can be realised by letting the operating system synch its file cache automatically to disk when it needs to. If you are using asynchronous writes, carefully consider your backup strategy, just in case the worst happens.

The pre InterBase V6.0 Server Manager offered some database validation capabilities, as does IBConsole, however I would recommend using the command line gfix utility to repair database corruption, it has more options and flexibility.

Database corruption that can be repaired can most often be corrected either by gfix, or a combination of gfix and gbak.

1. Define the following two variables, it makes life easier, in that you do not have to type in the user name and password every time you issue a command.

```
SET ISC_USER=SYSDBA
SET ISC_PASSWORD=masterkey
```

2. Always make sure you work on a copy of the database, not the production database. Use the operating system to make a copy of the database. You must have exclusive access to the database to do this.

```
copy employee.gdb database.gdb
```

3. Now check for database corruption. You must have exclusive access to do this, but since you're working on a copy of the original database, this is not a problem.

```
gfix –v –full database.gdb
```

4. If the previous command has indicated that there are problems with the database, we now need to mend it.

```
gfix –mend –full –ignore database.gdb
```

5. Now check to see if the corruption has been repaired.

```
gfix –v –full database.gdb
```

6. If you still see errors, you should now do a full backup and restore. In its simplest format the backup command line should be:

```
gbak –backup –v –ignore database.gdb database.gbk
```

7. However if gbak falls over because it is having trouble with garbage collection, then use the following command:

```
gbak –backup –v –ignore –garbage
database.gdb database.gbk
```

8. If there is corruption in record versions of a limbo transaction, then you may need to include the -limbo switch:

```
gbak –backup –v –ignore –garbage –limbo
database.gdb database.gbk
```

9. Now create a new database from the backup:

```
gbak –create –v atlas.gbk atlas_new.gdb
```

10. If there are problems on the restore, consider using the following switches.

-inactive, if there are index problems, this will restore the database, but will not activate any indexes, you can then do this manually, one at a time.

-one_at_a_time, this will restore the database one table at a time, and commit the restored tables on each table restore, if there is a major problem, at least you can get some of the data back.

If the above does not work, but you can still access the corrupt database, consider using QLI to move the data and table structures from the damaged database to a newly created one.

1. Create an empty database.

2. Edit the following (get_tables.sql) to point at the corrupt database.

```
connect database.gdb user 'sysdba' password 'masterkey;

select 'define relation tgt.', rdb$relation_name,
```

```
 ' based on relation src.',  rdb$relation_name, ';'
from rdb$relations where rdb$relation_name
not starting with 'RDB$';
commit;

select 'tgt.', rdb$relation_name, ' = src.',
rdb$relation_name, ';'
from rdb$relations where rdb$relation_name
not starting with 'RDB$';
```

3. Edit the output file so it looks something like this:

```
ready old.gdb as src;
ready new.gdb as tgt;

define relation tgt.COUNTRY
based on relation src.COUNTRY;
define relation tgt.JOB
based on relation src.JOB;
define relation tgt.DEPARTMENT
based on relation src.DEPARTMENT;
define relation tgt.EMPLOYEE
based on relation src.EMPLOYEE;
define relation tgt.PROJECT
based on relation src.PROJECT;
define relation tgt.PHONE_LIST
based on relation src.PHONE_LIST;
define relation tgt.EMPLOYEE_PROJECT
based on relation src.EMPLOYEE_PROJECT;
define relation tgt.CUSTOMER
based on relation src.CUSTOMER;
define relation tgt.SALES
based on relation src.SALES;
define relation tgt.PROJ_DEPT_BUDGET
based on relation src.PROJ_DEPT_BUDGET;
define relation tgt.SALARY_HISTORY
based on relation src.SALARY_HISTORY;

tgt.COUNTRY              = src.COUNTRY;
tgt.JOB                  = src.JOB;
tgt.DEPARTMENT           = src.DEPARTMENT;
tgt.EMPLOYEE             = src.EMPLOYEE;
tgt.PROJECT              = src.PROJECT;
tgt.PHONE_LIST           = src.PHONE_LIST;
tgt.EMPLOYEE_PROJECT     = src.EMPLOYEE_PROJECT;
tgt.CUSTOMER             = src.CUSTOMER;
tgt.SALES                = src.SALES;
tgt.PROJ_DEPT_BUDGET     = src.PROJ_DEPT_BUDGET;
tgt.SALARY_HISTORY       = src.SALARY_HISTORY;
```

4. Now install the appropriate version of QLI in the interbasebin directory and run the QLI script, by invoking QLI and running the move.sql script.

```
QLI>@move.sql
```

To help you understand exactly what gfix is doing, here is an extract from the source code, that explains in detail what is happening.

```
/*
 *      PROGRAM:        JRD Access Method
 *      MODULE:         val.c
 *      DESCRIPTION:    Validation and garbage collection
 *
 * copyright (c) 1985, 1997 by Borland International
 * copyright (c) 1999 by Inprise Corporation
 */

#ifdef INTERNAL_DOCUMENTATION
Database Validation and Repair
==============================

Deej Bredenberg March 16, 1994
Updated: 1996-Dec-11 David Schnepper

I. TERMINOLOGY

The following terminology will be helpful to understand in
this discussion:

record fragment:The smallest recognizable piece of a record;
multiple fragments can be linked together to form a single
version.
record version: A single version of a record representing an
INSERT, UPDATE or DELETE by a particular transaction (note
that deletion of a record causes a new version to be stored as a
deleted stub).
record chain: A linked list of record versions chained together
to represent a single logical "record".
slot: The line number of the record on page.
A variable-length array on each data page stores the offsets
to the stored records on that page, and the slot is an index
into that array.

II. COMMAND OPTIONS

Here are all the options for gfix which have to do with
validation, and what they do:

gfix switch   dpb parameter
-----------   -------------

-validate    isc_dpb_verify  (gds__dpb_verify prior to 4.0)

Invoke validation and repair.  All other switches modify this
switch.

-full        isc_dpb_records

Visit all records.  Without this switch, only page structures
will be validated, which does involve some limited checking of
records.

-mend        isc_dpb_repair

Attempts to mend the database where it can to make it viable
```

for reading; does not guarantee to retain data.

-no_update   isc_dpb_no_update

Specifies that orphan pages not be released, and allocated pages not be marked in use when found to be free. Actually a misleading switch name since -mend will update the database, but if -mend is not specified and -no_update is specified, then no updates will occur to the database.

-ignore   isc_dpb_ignore

Tells the engine to ignore checksums in fetching pages. Validate will report on the checksums, however. Should probably not even be a switch, it should just always be in effect. Otherwise checksums will disrupt the validation. Customers should be advised to always use it.
NOTE: Unix 4.0 (ODS 8.0) does not have on-page checksums, and all platforms under ODS 9.0 do not have checksums.

III. OPERATION

Validation runs only with exclusive access to the database, to ensure that database structures are not modified during validation. On attach, validate attempts to obtain an exclusive lock on the database.

If other attachments are already made locally or through the same multi- client server, validate gives up with the message:

"Lock timeout during wait transaction
-- Object "database_filename.gdb" is in use"

If other processes or servers are attached to the database, validate waits for the exclusive lock on the database (i.e. waits for every other server to get out of the database).

NOTE: Ordinarily when processes gain exclusive access to the database, all active transactions are marked as dead on the Transaction Inventory Pages. This feature is turned off for validation.

IV. PHASES OF VALIDATION

There are two phases to the validation, the first of which is a walk through the entire database (described below). During this phase, all pages visited are stored in a bitmap for later use during the garbage collection phase.

A. Visiting Pages

During the walk-through phase, any page that is fetched goes through a basic validation:

1. Page Type Check

Each page is check against its expected type. If the wrong type page is found in the page header, the message:

"Page xxx wrong type (expected xxx encountered xxx)"

is returned. This could represent a) a problem with the database being overwritten, b) a bug with InterBase page allocation mechanisms in which one page was written over another, or c) a page which was allocated but never written to disk (most likely if the encountered page type was 0).

The error does not tell you what page types are what, so here they are for reference:

```
#define pag_undefined     0    /* purposely undefined */
#define pag_header        1    /* Database header page */
#define pag_pages         2    /* Page inventory page */
#define pag_transactions  3    /* Transaction inventory page */
#define pag_pointer       4    /* Pointer page */
#define pag_data          5    /* Data page */
#define pag_root          6    /* Index root page */
#define pag_index         7    /* Index (B-tree) page */
#define pag_blob          8    /* Blob data page */
#define pag_ids           9    /* Gen-ids */
#define pag_log           10   /* Write ahead log page: 4.0 only */
```

2. Checksum

If -ignore is specified, the checksum is specifically checked in validate instead of in the engine. If the checksum is found to be wrong, the error:

"Checksum error on page xxx"

is returned. This is harmless when found by validate, and the page will still continue to be validated - if data structures can be validated on page, they will be. If -mend is specified, the page will be marked for write, so that when the page is written to disk at the end of validation the checksum will automatically be recalculated.

Note: For 4.0 only Windows & NLM platforms keep page checksums.

3. Revisit

We check each page fetched against the page bitmap to make sure we have not visited already. If we have, the error:

"Page xxx doubly allocated"

is returned. This should catch the case when a page of the same type is allocated for two different purposes.

Data pages are not checked with the Revisit mechanism - when walking record chains and fragments they are frequently revisited.

B. Garbage Collection

During this phase, the Page Inventory (PIP) pages are checked against the
bitmap of pages visited. Two types of errors can be detected during
this phase.

1. Orphan Pages

If any pages in the page inventory were not visited
during validation, the following error will be returned:

"Page xxx is an orphan"

If -no_update was not specified, the page will be marked as free
on the PIP.

2. Improperly Freed Pages

If any pages marked free in the page inventory were in fact
found to be in use during validation, the following error
will be returned:

"Page xxx is use but marked free" (sic)

If -no_update was not specified, the page will be marked in use
on the PIP.

NOTE: If errors were found during the validation phase, no changes
will
be made to the PIP pages. This assumes that we did not have a chance
to
visit all the pages because invalid structures were detected.

V. WALK-THROUGH PHASE

A. Page Fetching

In order to ensure that all pages are fetched during validation, the
following pages are fetched just for the most basic validation:

1. The header page (and for 4.0 any overflow header pages).
2. Log pages for after-image journalling (4.0 only).
3. Page Inventory pages.
4. Transaction Inventory pages

If the system relation RDB$PAGES could not be read or did not
contain any TIP pages, the message:

"Transaction inventory pages lost"

will be returned. If a particular page is missing from the
sequence as established by RDB$PAGE_SEQUENCE, then the following
message will be returned:

"Transaction inventory page lost, sequence xxx"

If -mend is specified, then a new TIP will be allocated on disk and

stored in RDB$PAGES in the proper sequence. All transactions which would have been on that page are assumed committed.

If a TIP page does not point to the next one in sequence, the following message will be returned:

"Transaction inventory pages confused, sequence xxx"

5. Generator pages as identified in RDB$PAGES.

B. Relation Walking

All the relations in the database are walked. For each relation, all indices defined on the relation are fetched, and all pointer and data pages associated with the relation are fetched (see below).

But first, the metadata is scanned from RDB$RELATIONS to fetch the format of the relation. If this information is missing or corrupted the relation cannot be walked.
If any bugchecks are encountered from the scan, the following message is returned:

"bugcheck during scan of table xxx (<table_name>)"

This will prevent any further validation of the relation.

NOTE: For views, the metadata is scanned but nothing further is done.

C. Index Walking

Prior to 5.0 Indices were walked before data pages.
In 5.0 Index walking was moved to after data page walking.
Please refer to the later section entitled "Index Walking".

D. Pointer Pages

All the pointer pages for the relation are walked. As they are walked all child data pages are walked (see below). If a pointer page cannot be found, the following message is returned:

"Pointer page (sequence xxx) lost"

If the pointer page is not part of the relation we expected or if it is not marked as being in the proper sequence, the following message is returned:

"Pointer page xxx is inconsistent"

If each pointer page does not point to the next pointer page as stored in the RDB$PAGE_SEQUENCE field in RDB$PAGES, the following error is returned:

"Pointer page (sequence xxx) inconsistent"

E. Data Pages

Each of the data pages referenced by the pointer page is fetched.

If any are found to be corrupt at the page level, and -mend is specified, the page is deleted from its pointer page. This will cause a whole page of data to be lost.

The data page is corrupt at the page level if it is not marked as part of the current relation, or if it is not marked as being in the proper sequence. If either of these conditions occurs, the following error is returned:

"Data page xxx (sequence xxx) is confused"

F. Slot Validation

Each of the slots on the data page is looked at, up to the count of records stored on page. If the slot is non-zero, the record fragment at the specified offset is retrieved. If the record begins before the end of the slots array, or continues off the end of the page, the following error is returned:

"Data page xxx (sequence xxx), line xxx is bad"

where "line" means the slot number.

NOTE: If this condition is encountered, the data page is considered corrupt at the page level (and thus will be removed from its pointer page if -mend is specified).

G. Record Validation

The record at each slot is looked at for basic validation, regardless of whether -full is specified or not. The fragment could be any of the following:

1. Back Version

If the fragment is marked as a back version, then it is skipped. It will be fetched as part of its record.

2. Corrupt

If the fragment is determined to be corrupt for any reason, and -mend is specified, then the record header is marked as damaged.

3. Damaged

If the fragment is marked damaged already from a previous visit or a previous validation, the following error is returned:

"Record xxx is marked as damaged"

where xxx is the record number.

4. Bad Transaction

If the record is marked with a transaction id greater than the last transaction started in the database, the following error is returned:

"Record xxx has bad transaction xxx"

H. Record Walking

If -full is specified, and the fragment is the first fragment in a logical
record, then the record at this slot number is fully retrieved. This involves retrieving all versions, and all fragments of each particular version. In other words, the entire logical record will be retrieved.

1. Back Versions

If there are any back versions, they are visited at this point. If the back version is on another page, the page is fetched but not validated since it will be walked separately.

If the slot number of the back version is greater than the max records on page, or there is no record stored at that slot number, or it is a blob record, or it is a record fragment, or the fragment itself is invalid, the following error
message is returned:

"Chain for record xxx is broken"

2. Incomplete

If the record header is marked as incomplete, it means that there are additional fragments to be fetched--the record was too large to be stored in one slot.
A pointer is stored in the record to the next fragment in the list.

For fragmented records, all fragments are fetched to form a full record version. If any of the fragments is not in a valid position, or is not the correct length, the following error is returned:

"Fragmented record xxx is corrupt"

Once the full record has been retrieved, the length of the format is checked against the expected format stored in RDB$FORMATS (the format number is stored with the record, representing the exact format of the relation at the time the record was stored.)
If the length of the reconstructed record does not match
the expected format length, the following error is returned:

"Record xxx is wrong length"

For delta records (record versions which represent updates to the record)
this check is not made.

I. Blob Walking

If the slot on the data page points to a blob record, then the blob is fetched (even without -full). This has several cases, corresponding to the various blob levels.

Level	Action
0	These are just records on page, and no further validation is done.
1	All the pages pointed to by the blob record are fetched and validated in sequence.
2	All pages pointed to by the blob pointer pages are fetched and validated.
3	The blob page is itself a blob pointer page; all its children are fetched and validated.

For each blob page found, some further validation is done. If the page does not point back to the lead page, the following error is returned:

"Warning: blob xxx appears inconsistent"

where xxx corresponds to the blob record number. If any of the blob pages
are not marked in the sequence we expect them to be in, the following error is returned:

"Blob xxx is corrupt"

Tip: the message for the same error in level 2 or 3 blobs is slightly different:

"Blob xxx corrupt"

If we have lost any of the blob pages in the sequence, the following error
is returned:

"Blob xxx is truncated"

If the fetched blob is determined to be corrupt for any of the above reasons, and -mend is specified, then the blob record is marked as damaged.

J. Index Walking

In 5.0 Index walking was moved to after the completion
of data page walking.

The indices for the relation are walked. If the index root page is missing, the following message is returned:

"Missing index root page"

and the indices are not walked. Otherwise the index root page is fetched and all indices on the page fetched.
For each index, the btree pages are fetched from top-down, left to right.
Basic validation is made on non-leaf pages to ensure that each node on page points to another index page. If -full validation is specified then the lower level page is fetched to ensure it is starting index

entry is consistent with the parent entry.
On leaf pages, the records pointed to by the index pages are not fetched, the keys are looked at to ensure they are in correct ascending order.

If a visited page is not part of the specified relation and index, the following error is returned:

"Index xxx is corrupt at page xxx"

If there are orphan child pages, i.e. a child page does not have its entry
as yet in the parent page, however the child's left sibling page has it's
btr_sibling updated, the following error is returned

"Index xxx has orphan child page at page xxx"

If the page does not contain the number of nodes we would have expected from its marked length, the following error is returned:

"Index xxx is corrupt on page xxx"

While we are walking leaf pages, we keep a bitmap of all record numbers seen in the index. At the conclusion of the index walk we compare this bitmap to the bitmap of all records in the relation (calculated during data page/Record Validation phase). If the bitmaps are not equal then we have a corrupt index and the following error is reported:

"Index %d is corrupt (missing entries)"

We do NOT check that each version of each record has a valid index entry - nor do we check that the stored key for each item in the index corresponds to a version of the specified record.

K. Relation Checking

We count the number of backversions seen while walking pointer pages, and separately count the number of backversions seen while walking record chains. If these numbers do not match it indicates either "orphan" backversion chains or double-linked chains. If this is see the following error is returned:

"Relation has %ld orphan backversions (%ld in use)"

Currently we do not try to correct this condition, mearly report it. For "orphan" backversions the space can be reclaimed by a backup/restore. For double-linked chains a SWEEP should remove all the backversions.

VI. ADDITIONAL NOTES

A. Damaged Records

If any corruption of a record fragment is seen during validation, the record header is marked as "damaged". As far as I can see, this has no

effect on the engine per se. Records marked as damaged will still be retrieved by the engine itself. There is some question in my mind as to whether this record should be retrieved at all during a gbak.

If a damaged record is visited, the following error message will appear:

"Record xxx is marked as damaged"

Note that when a damaged record is first detected, this message is not actually printed. The record is simply marked as damaged. It is only thereafter when the record is visited that this message will appear. So I would postulate that unless a full validation is done at some point,
you would not see this error message; once the full validation is done, the message will be returned even if you do not specify -full.

B. Damaged Blobs

Blob records marked as damaged cannot be opened and will not be deleted from disk. This means that even during backup the blob structures marked as damaged will not be fetched and backed up. (Why this is done differently for blobs than for records I cannot say.
Perhaps it was viewed as too difficult to try to retrieve a damaged blob.)

#endif /* INTERNAL_DOCUMENTATION