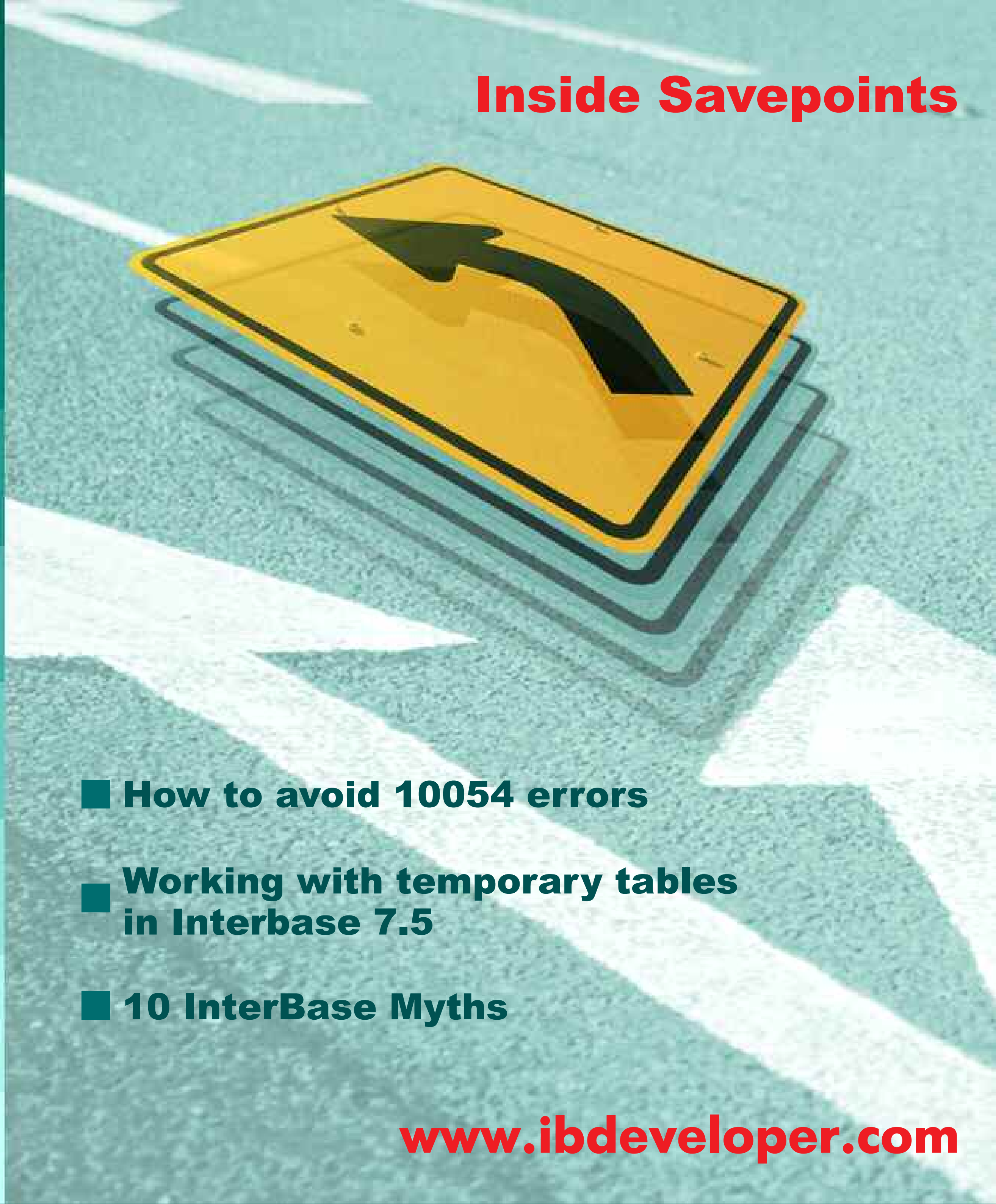
The InterBase and Firebird Developer Magazine



Inside Savepoints

- How to avoid 10054 errors
- Working with temporary tables in Interbase 7.5
- 10 InterBase Myths

#1

| 2005 | Issue 1 |

www.ibdeveloper.com

Contents

Subscribe now!
To receive future issues notifications send email to
subscribe@ibdeveloper.com

Credits
Alexey Kovyazin, **Chief Editor**
Dmitri Kouzmenko, **Editor**
Lev Tashchilin, **Designer**
Volker Rehn, **Subeditor**

Such a nice newborn

Editor's note

Dear colleagues!

I am happy to present to you the first issue of "The InterBase and Firebird Developer Magazine". It has been a long time since the initial idea for such a magazine was born, and now you are reading the first issue.

Why "InterBase and Firebird"

I think the "editor's note" is the right place to answer some obvious questions about a newborn magazine. First of all, why does its title contain both "InterBase" and "Firebird"? Some newbies and gurus of InterBase and Firebird insist that they are different products – and I completely agree. They are different, but people who use them are the same. Development approaches, problems and techniques are almost always the same.

Of course, InterBase and Firebird are different but it is not the same kind of difference as between Oracle and MS SQL – they are different like brothers.

Who need our magazine

You sometimes hear the opinion that "InterBase is not a serious DBMS" and "Firebird? What's this?" And it is a hardly known fact that the size of our community is close to the community size of such monsters as Oracle and MS SQL. Despite the fact that Oracle and Microsoft command huge PR-budgets, we are very well positioned in this competition. And I am sure that there is huge hidden potential in our community.

Now the situation is changing. We have got at least one great book about Firebird, we have annual conference events, and now we have got our own magazine – just like other big players.

I hope that our magazine will be useful and interesting for all people who work with InterBase and Firebird – as DBAs, as developers or even as project managers. We are concentrating on "all times greatest hits" problems and will try to make this magazine not a one-off reading, but a long-play interesting novel that makes for some good reaing a number of times. Also in future issues we will pay attention to community events like BorCon and the Firebird Conference.

In this issue

Ok, what do we have in this issue? The first and largest article "Inside Savepoints" by Dmitri Yemanov is devote to the internal details of server savepoints. Explicit savepoints are a rather recent improvement in both InterBase and Firebird, and I think it will be very interesting for all readers to know nitty-gritty details of their functioning.

Another hot article is "Using Embedded User Authentication in InterBase 7.5" by Dmitri Kouzmenko. We have all been waiting for this feature for a long time, and I suppose we won't be disappointed.

"Working with temporary tables in InterBase 7.5" is also very interesting. To my mind, adding user temporary tables is the most important recent improvement of InterBase so every developer should know its details.

And I think many InterBase and Firebird developers came across the most annoying error – error 10054. The article of Vasily Ovchinnikov is devoted to practical ways to avoid 10054 errors.

Of course, this is not all of it in this issue – please feel invited to read on.

We need your feedback
This is the first issue of "The Interbase and Firebird Developer Magazine". We have got plenty of ideas for the following issues and will try to make "The Interbase and Firebird Developer Magazine" the best database developer magazine around. And the most important thing we would like to know is your opinion and your thoughts about our work. Please do not hesitate to contact us:

readers@ibdeveloper.com

Sincerely yours, Alexey Kovyazin.

Chief Editor

Inside Savepoints

Author: Dmitri Yemanov
dimitr@users.sourceforge.net

General information

A savepoint is an internal mechanism of the database, which binds any changes in the database to a specific point of time during a transaction, and in case of necessity, allows a user to cancel all changes, which were made after setting this particular savepoint. This process is also known as rolling back to savepoint.

Also server uses savepoint mechanism to implement transaction handling This mechanism helps either to commit or to cancel all changes made during a transaction. For those purposes, the server uses the global savepoint.

Such a savepoint is set automatically when a transaction starts, and it is the first savepoint in the transaction context. When transaction rollback is initiated, all changes made within its context are cancelled using the transaction global savepoint. After that, the transaction is commited (!) within the Transaction Inventory Page (TIP). This is necessary in order to avoid housekeeping operations in the future.

However, if the number of changes in transaction context becomes too big (approx.10000 - 1000000 records), then the storing of rollback lists becomes expensive, and the server deletes the transaction global savepoint, switching to the standard TIP mechanism to mark the transaction as dead.

Tip:
If you expect that during transaction many changes are to be made, then it makes sense to specify the isc_tpb_no_auto_undo transaction parameter, which disables usage of the global savepoint for rollback. In some cases, it allows to increase server's performance during batch operations.

In addition to the use of savepoints for rollbacks, the server also uses them for exception handling. Each SQL and/or PSQL operator is enclosed in a savepoint frame, which allows to rollback this particular operator, keeping the previous ones unchanged. This guarantees either successful execution of the operator or automatic cancellation of all changes made, and a corresponding error will be initiated.

For exception handling in PSQL, each BEGIN...END block is also enclosed in a savepoint frame, which allows you to cancel all changes made by this block. Let's consider some details of how savepoints work.

Savepoints in action

A savepoint is a data structure, which is located in the server's dynamic storage (transaction pool) and has a unique numerical ID. A list of activities made within the savepoint context is associated with this savepoint. Such a list is called an "undo log." Savepoints form a stack within a transaction, and that is the reason why only sequential rollback of savepoints is possible.

Undo log fragments are distributed across savepoints that store the history. A savepoint, which is active when a record is being changed, is called "current". Information about record changes is stored in the current savepoint undo log. If a rollback to the savepoint is performed, the undo log is unwound, and records are reconstructed. As a result, the record becomes as it was at the moment this savepoint was set.

In case there are no exception handlers available, the records may be reconstructed down to the global savepoint, providing complete transaction rollback. After reconstruction of all modified records, the savepoint is usually deleted from the transaction context.

Releasing savepoints

In addition to the rollback to savepoint operation, there is an operation of regular deletion (release) of a savepoint. In case a savepoint is deleted, its undo log is merged with the undo log of the previous one, in the savepoint stack. In this sense, each savepoint is nested.

It is clear, that regular deletion of all savepoints located "deeper" than the global one, would lead to the transfer of all changes to the transaction global savepoint.

Figure 1 Savepoints in action

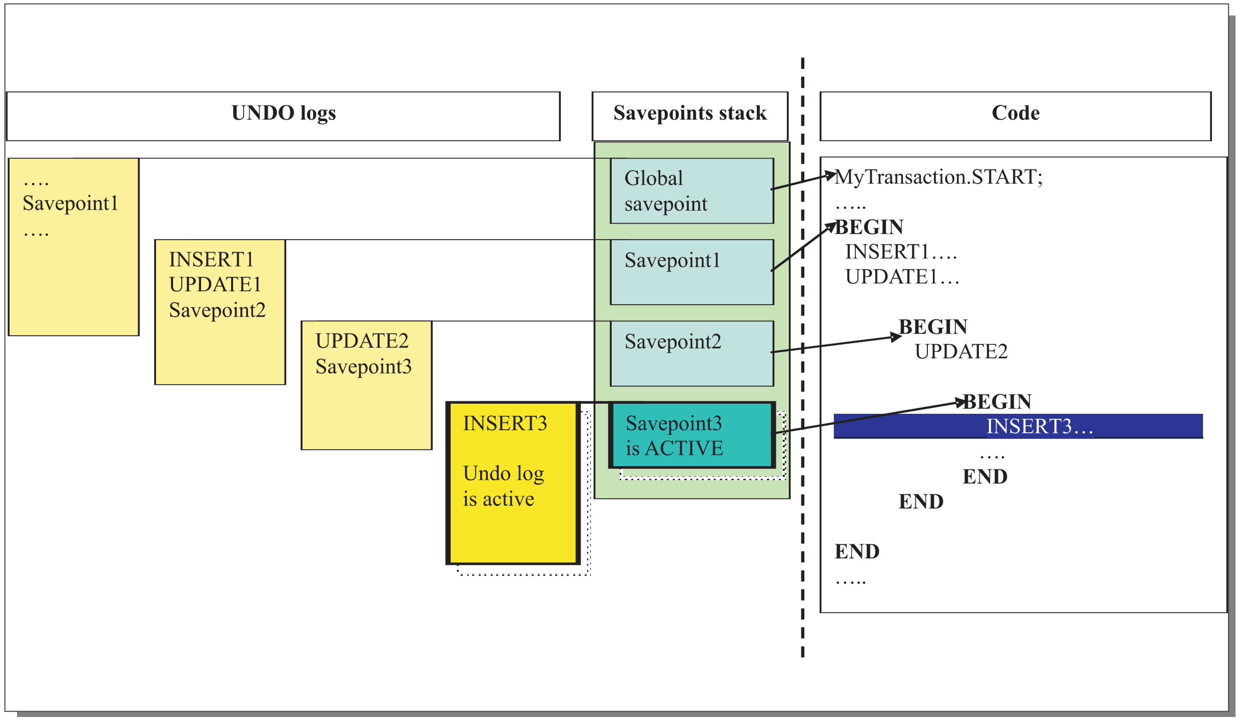
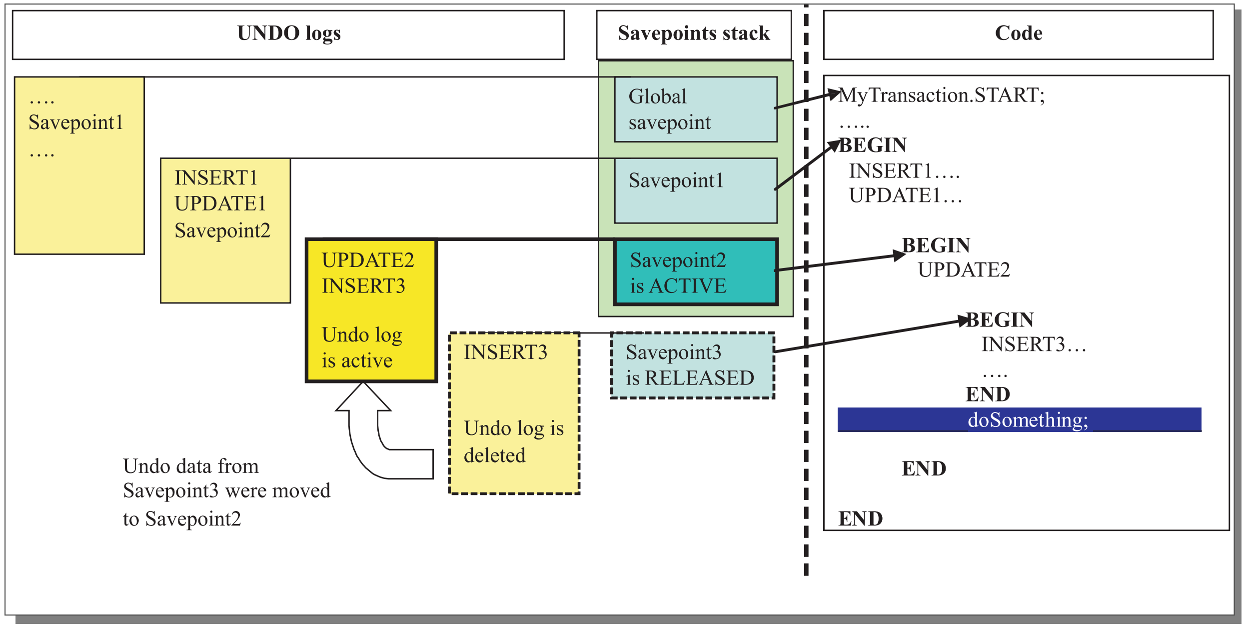


Figure 2 Releasing a savepoint: undo data moved to upward savepoint



Tip:
As is illustrated below, each custom SQL-query is a set of savepoints within a transaction.

Thus, the combination of all changes, which were successfully executed during transaction, is stored in undo logs. That is why, when the automatic undo log is enabled, the server can cancel all performed changes, and in the TIP this transaction would be marked as committed, and not as dead.

When specifying the *isc_tpb_no_auto_undo* parameter at transaction start, a global savepoint is not created, and if regular deletion of the current stack is performed, the combined undo log is simply deleted, and transaction is marked dead, and all changes are to be cleared (garbage collection).

Savepoints and exception handling

There are several events, which make server create system-defined (i.e. user-uncontrollable) savepoints:

1. Execution of any user SQL-query. As has been said above, this done in order to guarantee atomicity of a query. That is to say, if any exception occurs during query execution, the changes made in the database will always be canceled. After the query is executed, the savepoint will be automatically deleted.

2. Execution of the BEGIN...END block in PSQL (stored procedure or a trigger) in case the block contains an error handler (WHEN-block). In that case, each BEGIN operator sets a savepoint, and a corresponding END operator deletes it. This enables to provide error handling

in the PSQL-block.

3. Execution of an SQL-operator in the context of a "BEGIN...END" block, which contains an error handler (WHEN-block). That is, if the block contains an error handler, any SQL-operator in this block is framed by a savepoint frame.

Tip:
It is necessary to note that the *isc_tpb_no_auto_undo* parameter does not disable the savepoints mechanism (this is impossible due to the atomicity guarantee for SQL-operators). It only disables the creation of a transaction undo log as a single whole.

Exception handling

Let's consider the errors handling process on the server. When an exception occurs, automatic rollback to the last set savepoint is performed. As a result, all operations performed by an invalid SQL-query would be canceled.

Then, in case of a PSQL-block, it is checked, if there is a custom WHEN-handler. If it does exist, control is transferred to it, and after exiting, savepoint is deleted.

Then the process repeats recursively, until embedded handlers end. Thus, if a stored procedure doesn't contain a handler which would be able to handle this error, the whole procedure will be considered as a single SQL-operator, and canceled.

Let's make a summary

Firstly, if there is no WHEN-handler, any PSQL-block (including stored procedure and trigger) becomes atomic, and would be canceled entirely, if an error occurs.

Secondly, in the presence of a WHEN-handler, the occurrence of an error leads a rollback of the only operator, and after that the process is managed by the handler.

That is to say, there are obvious differences in the server's reactions, which depend on the presence of a WHEN-handler.

It is worth considering a known anomaly, which does not fit the scheme described above. The scheme works as follows: the paragraph 3 (enclosing of each SQL-operator inside a PSQL-block by its own savepoint frame) is true for SQL-operators only (moreover, not for all of them). In other words, for example, an assignment operator will not be framed by a savepoint frame.

As a result of an error in assignment, execution leads to a normal rollback to the previous savepoint, which is... exactly! – the block's savepoint.

That is to say, even if there is a WHEN-handler, the error may cause the rollback of the whole block before control to the error handler.

I consider this situation as a serious flaw in the server's exceptions handling logic. Below is the full list of operators, for which the savepoint frame is created: INSERT, UPDATE, DELETE, EXCEPTION, EXECUTE STATEMENT [INTO].

To understand the reasons of this anomaly, it is necessary to take into account the following 2 facts:

• If an error occurs, rollback to the last savepoint is performed unconditionally.

• A block's savepoint deletion (or rollback to it) is performed together with the verification of the savepoint's identifier

This means that when an error occurs in operators not included to the above list (i.e. not enclosed in a savepoint frame), that actually the wrong (block's) savepoint is deleted. But the block itself is tolerant to that fact, since it only deletes its own savepoint (in case it has one). This is the reason why the described server error does not cause fatal consequences.

Let us illustrate such a situation, using the description of the [428903] Exception Handling Bug error. To clarify this, we provide these examples with comments about show the server deals with savepoints.

Example 1
A procedure with an error handler and error generation in the assignment operator:

```
CREATE PROCEDURE PROC1
AS
  DECLARE VARIABLE X INT;
-- start savepoint #1
BEGIN
  -- start savepoint #2
  INSERT INTO TAB (COL) VALUES
(01);
  -- end savepoint #2
  X = 1 / 0;
WHEN ANY DO
  EXIT;
-- end savepoint #1
END
```

In this case, since savepoint #1 is the nearest to the erratic operator (savepoint #2 was deleted right before assignment execution), INSERT will be canceled. Therefore, a rollback of the whole BEGIN...END block will be performed before entering the handler.

Below is the same procedure with an explicit exception statement, enclosed in a block without a WHEN-handler:

```
CREATE PROCEDURE PROC2
AS
  DECLARE VARIABLE X INT;
-- start savepoint #1
BEGIN
  -- start savepoint #2
  INSERT INTO TAB (COL) VALUES (23);
  -- end savepoint #2
  BEGIN
    X = 1 / 0;
  END
WHEN ANY DO
  EXIT;
-- end savepoint #1
END
```

As you see, no savepoint frame near assignment operator was created. Therefore, the result would be similar to the previous one.

Below is the same procedure with an explicit exception call, in the block, which contains WHEN-handler:

```
CREATE PROCEDURE PROC3
AS
  DECLARE VARIABLE X INT;
-- start savepoint #1
BEGIN
  -- start savepoint #2
  INSERT INTO TAB (COL) VALUES (45);
  -- end savepoint #2
  -- start savepoint #3
  BEGIN
    X = 1 / 0;
  WHEN ANY DO
    EXIT;
  -- end savepoint #3
  END
WHEN ANY DO
  EXIT;
-- end savepoint #1
END
```

Here we see a created savepoint frame. As a result, rollback is performed only for the nearest BEGIN level (savepoint #3), and the INSERT operator remained executed.

InterBase Myths № 1

When performing a "restore" outdated, versions are deleted (and, therefore, are stored in backup) or gbak -g creates a backup file without versions, and by default versions are included in the file.

Nothing of the kind! No records versions are stored in a backup because it is unnecessary. Generally, the backup process is an ordinary snapshot transaction (repeatable read), and it reads only those records versions, which were relevant at the beginning of the transaction. The "no_garbage_collect" flag controls collecting garbage versions in the database itself. This flag can also be used during ordinary connections accessing the AP (i.e. in applications, when, say, one needs to accelerate sampling in some cases).

Example 2
A procedure with error handler and explicit exception call:

```
CREATE PROCEDURE PROC4
AS
-- start savepoint #1
BEGIN
  -- start savepoint #2
  INSERT INTO TAB (COL) VALUES (67);
  -- end savepoint #2
  -- start savepoint #3
  EXCEPTION E;
  -- end savepoint #3
WHEN ANY DO
  EXIT;
-- end savepoint #1
END
```

In this case, INSERT will not be canceled, due to the fact that the exception initiation of E results in rollback to savepoint #3 and subsequent transfer of control to the error handler. In order to cancel the INSERT operator in this case, you should inhibit the execution of the deletion handler:

```
CREATE PROCEDURE PROC5
AS
BEGIN
  INSERT INTO TAB (COL) VALUES (89);
  EXCEPTION E;
END
```

In addition, a few words about exception handling adequacy regarding the SQL-standard.

The standard allows three types of handlers in PSQL: **CONTINUE**, **EXIT**, and **UNDO**. With a **CONTINUE**-handler, the server must rollback the erroneous operator, execute the handler code, and then continue the execution of the block, beginning with the operator next to the which one caused the error.

An EXIT-handler requires finishing of the execution of the block right after exiting the handler code.

An UNDO-handler requires a rollback of all actions of the block before entering the handler.

Current versions of the server (InterBase as well as Firebird) do not support the explicit specification of the handler type, and work according to the EXIT principle (however, there is a possibility of UNDO-behavior due to the anomaly described above).

I suppose that in the future, it would be desirable to provide an alternative to choose between UNDO- and EXIT- behavior of a handler, and repair the described anomaly.

Custom savepoints

In addition to the internal realization of savepoints at transaction (and operator/block) levels, the latest versions of servers (InterBase 7.1, Firebird 1.5, and Yaffil 1.1) provide an SQL-interface, developed for this mechanism.

Note: *savepoints' syntax and semantics are declared in the SQL-99 standard (see section 4.37.1 of the specification).*

Custom savepoints (also known as nested transactions) provide a convenient business logic error handling method, with no need to rollback the whole transaction.

Note: *rollback to a savepoint is also sometimes called "partial transaction rollback."*

New SQL operator (SAVEPOINT) was added to define a savepoint in the transaction context, to which a rollback can be performed later on:

```
SAVEPOINT <name>;
```

<name> - the string identifier of a savepoint. As soon as a savepoint is created, you can either continue transaction, commit (or cancel) the whole transaction, or perform a rollback to a particular savepoint. Savepoints' names (identifiers) must be unique in the context of a transaction. If you attempt to create two savepoints with similar names, the first savepoint is deleted, and the specified name is given to the second one.

For rollback to a savepoint, the following operator is used:

```
ROLLBACK [ WORK]  TO [ SAVEPOINT]  <name>;
```

Note: *the SAVEPOINT keyword is obligatory in InterBase 7.1.*

During execution of this operator, the following actions are performed:

• Rollback of all changes made after the savepoint was set;

• All savepoints set after this one are deleted. The current savepoint remains unchanged, and thus you can perform several rollbacks to a savepoint. Previous savepoints remain unchanged as well.

Note: *Performing a rollback to savepoint in InterBase 7.1 deletes the selected savepoint.*

• All explicit and implicit write locks, occupied after the savepoint was set, are released. At that, other transactions, which requested an access to the records blocked by the transaction after the savepoint was set, continue waiting for the current transaction to be finished. Transactions, which did not request access to the records, may continue and get access to them.

Note: *This behavior refers to Firebird 1.5 and can be changed in higher versions.*

Since each savepoint uses certain system resources, and also clogs the namespace, it makes sense to release (delete) savepoints when they are no longer necessary. This can be accomplished using the following operator:

```
RELEASE SAVEPOINT <name> [ ONLY] ;
```

This command deletes the selected (and all following) savepoints from transaction context. The "ONLY" option is a switch to delete the selected savepoint only; at that, all following savepoints will be saved. If a savepoint was not released explicitly, it will be automatically deleted as soon as the transaction is finished.

Note: *The "ONLY" option is non-standard extension, and is not supported by InterBase 7.1.*

A simple example of working with savepoints is given below:

```
create table test (id int);
commit;
insert into test (id) values (1);
commit;
insert into test (id) values (2);
savepoint y;
delete from test;
select * from test; -- returns empty set
rollback to y;
select * from test; -- returns two records
rollback;
select * from test; -- returns one record
```

A custom savepoint

Now let us consider an example of how savepoints can be used in business logic. Assume there is an operation of mass document handling in the application, and it is necessary to display error messages (or save them for future presentation as a list), and let this bulk operation continue. Since the document handling operation is not atomic, on the client's side it is better not to use regular exception handling since we cannot continue the transaction if we know that an exception performed a rollback of only half of the operation.

Such a dilemma can be resolved by handling each document sequentially in a separate transaction. Nevertheless, this does increase the consumption of internal server resources (maximum number of records in TIP, transaction counter increment), and is therefore not the best alternative.

In addition, if there is a need to fix a set of documents during the handling process (for example, by changing the transaction isolation mode or explicit blocking of the SELECT ... WITH LOCK type), it would require using only one transaction for the delta packet. Using a savepoint, the following algorithm would be used (in pseudocode):

```
START TRANSACTION;
OPEN C FOR ( SELECT … );
FOR ( C ) DO
  LOOP
    TRY
      SAVEPOINT DOC;
       <..> ///single document handling commands
    EXCEPT
      ROLLBACK TO SAVEPOINT DOC;
   <...> //either log the error or display it
    END
  END
CLOSE C;
COMMIT;
```

Note: *The use of savepoints in loops has an additional advantage: you do not need to call RELEASE each time, since resetting a savepoint automatically deletes the previous savepoint with the same name.*

Another example is, undoubtedly, audit. For example, you need to provide a log record for each activity, and at the same time, if an error occurs, the record should remain in the audit log (with a corresponding note):

```
START TRANSACTION;
INSERT INTO AUDIT_LOG (ID, EVENT, STATUS) VALUES (:ID, :EVENT, 1);
SAVEPOINT OPER;
TRY
 <...> // operations on database
EXCEPT
  ROLLBACK TO SAVEPOINT OPER;
  UPDATE AUDIT_LOG SET STATUS = 0 WHERE ID = :ID;
END
COMMIT;
```

InterBase Myths № 2

Record versions are created during reading

Versions are created only when records are changed or deleted (UPDATE or DELETE). On the contrary, if unnecessary versions of the same record are detected, they are marked as trash (i.e., they would be deleted). Read as much as you want, this will not result in creation of new versions. Vice versa, updating of a record creates a new version of the record in any case, disregarding the fact that someone else reads this record.

A myth sometimes becomes true: versions of the records may be created during reading, but only in Firebird 1.5 (and higher) in queries select ... for update with lock. More detailed information will be published in the next issue of full title.

Savepoints in stored procedures and triggers

Now let us consider the usage of custom savepoints in procedures and triggers.

At first glance, it looks very attractive and useful. Originally this functionality is declared in InterBase 7.1. Let's consider the capabilities in detail.

First of all, savepoints must not break the atomicity of SQL-operators. This means that none of the commands can be canceled partially. Remember that EXECUTE PROCEDURE is a legal SQL-operator, and any operators updates may lead to trigger execution. Generally speaking, any "simple" operator, such as INSERT or UPDATE may result in launching of a whole chain of triggers and procedures. That is why we are to examine the scope of a savepoint.

It is obvious that in order to meet the atomicity requirement, savepoint affected instructions with in a procedure should not have an access to the transaction savepoint (set through the SAVEPOINT global operator). In addition, procedure savepoints must be local and their scope must be defined by the procedure. That is to say, there can be a savepoint named "S1" both in the transaction and in procedures and triggers executed in the context of this transaction. At that, such savepoints will be isolated from each other. Note that this very method is used in InterBase 7.1.

A question emerges: how would custom savepoints coexist with internal savepoints managed by the server?

Some theory

Let us consider a simple example of savepoint usage in PSQL, suggested by Borland in the InterBase 7.1 server documentation:

```
CREATE PROCEDURE ADD_EMP_PROJ2 (
  EMP_NO SMALLINT,
  EMP_NAME VARCHAR(20),
  PROJ_ID CHAR(5) )
AS
BEGIN
  BEGIN
    SAVEPOINT EMP_PROJ_INSERT;
    INSERT INTO EMPLOYEE_PROJECT (EMP_NO, PROJ_ID)
 VALUES (:EMP_NO, :PROJ_ID);
  WHEN SQLCODE -530 DO
    BEGIN
      ROLLBACK TO SAVEPOINT EMP_PROJ_INSERT;
      EXCEPTION UNKNOWN_EMP_ID;
    END
  END
END
```

This example demonstrates how exceptional situations are handled when using savepoints. That is to say, when an exception with code -530 occurs (the violation of reference integrity by a foreign key) we cancel the insert operation and initiate a user exception. Actually, this example is absolutely useless, since we do not need a savepoint here:

```
BEGIN
  INSERT INTO …
WHEN SQLCODE -530 DO
  EXCEPTION unknown_emp_id;
END
```

This code will execute the same function, since the server itself cancels the INSERT operation when an exception during it's execution.

Let's consider a more complicated example:

```
FOR SELECT ID, … INTO :REC_ID, …
BEGIN
  SAVEPOINT S1;
  INSERT INTO TABLE1 …
  INSERT INTO TABLE2 …
  INSERT INTO TABLE3 …
  EXECUTE PROCEDURE …
  …
WHEN ANY DO
  BEGIN
    ROLLBACK TO SAVEPOINT
S1;
    ERROR = REC_ID;
    SUSPEND;
  END
END
```

Here we try to handle all documents, but the program does not stop in case of failure, it only returns all unsuccessful attempts at the end of the procedure. Server standard logic would cancel the error operator and control to the handler, which in its turn cancels actions of the whole block. Thus, we can easily turn the EXIT-handler to UNDO in case if necessary. Of course, this functionality can be obtained by standard means as well:

```
FOR SELECT ID, … INTO
:REC_ID, …
BEGIN
  BEGIN
    INSERT INTO TABLE1 …
    INSERT INTO TABLE2 …
    INSERT INTO TABLE3 …
    EXECUTE PROCEDURE …
    …
  END
WHEN ANY DO
  BEGIN
    ERROR = REC_ID;
    SUSPEND;
  END
END
```

In this case, all operators within the loop will be automatically canceled, in the event that an exception occurs, since the operators are located in the atomic block by which the savepoint frame for SQL-operators mechanism was enabled. After that, the server will go through the chain of embedded blocks, and will switch to the handler.

Therefore, in virtually any case, one can realize the same semantics using the server's standard mechanisms, i.e. using system savepoints instead of custom ones, at the cost of relatively unhandy source code. Thus, savepoints in PSQL are nothing but an easy and comprehensive alternative for the explicit usage of BEGIN…WHEN…END blocks.

A bit of practice

Now let us return from theory to practice and test this reasoning in InterBase 7.1. The result is quite depressing: none (!!!) of the given examples work, and error messages appear:

```
Statement failed, SQLCODE = -504
Savepoint <name> unknown.
```

Even the first example, which was taken from the Release Notes (!), is not working properly. At the same time, the most primitive examples, such as:

```
SAVEPOINT S1;
INSERT …
ROLLBACK TO SAVEPOINT S1;
```

work correctly. So what's the matter? If we investigate the situation more carefully, the reason becomes obvious. Remember the two facts described above:

1. savepoints constitute a stack, and can be canceled sequentially only

2. each block of PSQL-code with an exception handler is enclosed in a frame

Thus we arrive at a conclusion that any code area of the following type:

```
SAVEPOINT S1;
…
BEGIN
…
ROLLBACK TO SAVEPOINT S1;

…
WHEN
```

is definitely invalid, since to perform a rollback to savepoint S1, it would be necessary to delete the system savepoint, created by the server for exception handling in the "BEGIN…END" block. This would destroy the internal undo log, and may corrupt the database.

Thank God, the InterBase developers did not create such cardinal realization, and server attempts to cancel the previous (last) savepoint directly, only if its name matches. Since system savepoints are unnamed, in this case such an attempt would fail. This is proven by the above mentioned error message.The above makes us arrive to the conclusion that working with savepoints in PSQL is limited by the nesting level, in case we are dealing with blocks with a WHEN-handler.

However, it turned out that the most interesting thing is yet to come. The server's reaction to the error initiated by the ROLLBACK TO SAVEPOINT or RELEASE SAVEPOINT operator is amusing. Let's illustrate this using an example:

```
BEGIN
   INSERT INTO TABLE1 …
   ROLLBACK TO SAVEPOINT S1;
   INSERT INTO TABLE2 …
END
```

This is an emulation of an error, which usually occurs if the required savepoint cannot be found within a single code block. As one would expect, the execution of the procedure returns the same error. But!!! Procedure execution does not stop at this point. Instead, the second INSERT is executed (which you can easily verify by substituting INSERT with an operator of EXCEPTION E_TEST type). The question is, why? It turns out that this error cannot be handled in the procedure, i.e. the code:

```
INSERT INTO TABLE1 …
BEGIN
   ROLLBACK TO SAVEPOINT S1;
   WHEN ANY DO
      EXCEPTION E_TEST;
END
```

does not throw the E_TEST exception, as one might expect. Even though the code after ROLLBACK TO SAVEPOINT is executed, nothing really happens. Which means, that in case the described error occurs in a procedure, all changes made by this procedure will be unconditionally (!) canceled. This happens regardless of which code was executed before or after the command. It would be interesting to find out how InterBase developers explain this phenomenon.

Summary

There are some peculiarities in savepoints logic, which prevent realization of their complete support by PSQL. The analysis of InterBase 7.1 behavior proves the point. The rationale of that is the presence of system savepoints, which interaction with custom ones is limited, due to data integrity requirements. That's why this functionality is neither available in Firebird, nor in Yaffil.

Note: *As far as I understand it, the same reasons prevent from using commit/rollback retaining in PSQL, since in that case the savepoint-frame of a procedure would be destroyed.*

Savepoint in distributed transactions

InterBase 7.1 introduces the option to work with savepoints in coordinated transactions.

For this purpose, three new API functions are introduced:

```
ISC_STATUS isc_start_transaction(ISC_STATUS* status,
                                 isc_tr_handle* trans, char* name);
ISC_STATUS isc_release_transaction(ISC_STATUS* status,
                                 isc_tr_handle* trans, char* name);
ISC_STATUS isc_rollback_transaction(ISC_STATUS* status,
                                 isc_tr_handle* trans, char* name, short option);
```

As you see, these functions do not have a connection descriptor (database handle) which means that corresponding SQL-commands are issued to all databases used by the transaction. This seems absolutely logical, since formally, a savepoint is a part of a transaction and not of a connection. However, there is one nuance here. Let's examine following program fragment (error handling is not included):

```
/* Connect to the database */
isc_attach_database(status, 0, database1, &db1, 0, NULL);
isc_attach_database(status, 0, database2, &db2, 0, NULL);

/* Begin coordinated transaction */
isc_start_transaction(status, &trans, 2, &db1, 0, NULL, &db2, 0, NULL);

/* Create savepoint */
isc_start_savepoint(status, &trans, "A");

/* Executing database operations */
isc_dsql_execute_immediate(status, &db1, &trans, 0, "DELETE FROM TABLE1", 1, NULL);
isc_dsql_execute_immediate(status, &db2, &trans, 0, "DELETE FROM TABLE2", 1, NULL);
/* Delete the savepoint explicitly, through the second connection descriptor */
isc_dsql_execute_immediate(status, &db2, &trans, 0, "RELEASE SAVEPOINT A", 1, NULL);

/* Rollback to savepoint */
isc_rollback_savepoint(status, &trans, "A", 0);
```

```
/* Commit coordinated transaction */
isc_commit_transaction (status, &trans);

/* Disconnecting the database */
isc_detach_database(status, &db1);
isc_detach_database(status, &db2);
```

As a result of a rollback to savepoint, I expect rollback to be performed in both databases I work with. Then I commit the transaction, and after that I need to see all data on their places, since DELETE operators were canceled.

And that would be that way, but for the manually executed "RELEASE SAVEPOINT A."

At first, savepoint rollback was performed for the first connection and all changes were canceled.

Then the same operation was accomplished for second connection, while... oops! ... there is no savepoint anymore. As a result, the client receives an error message. But rollback of one of the DELETE operators was successful (!) This is a situation, when the coordinated operation disintegrates itself, and makes correct handling of the case impossible.

The two-phase fixation of transaction mechanism, which should bar from such cases, simply cannot deal with savepoints.

That is to say, the new InterBase 7.1 functions create appropriate SQL-commands, and then cyclically execute them for databases involved in this process.

Even if only one of them fails, an error returns. Generally, this error does not characterize the current situation, in terms of correctness of the operation as a single whole.

Of course, one might say that only one method of working with savepoints should be applied – either through SQL, or through API.

Now you know what it leads to. However, for working with savepoints in InterBase 7.1, a new API can be completely substituted by server's standard means.

The server should either suppress possibility to control savepoints in certain connections in cases of coordinated transactions, or refuse to declare their workability.

It is necessary to note that Firebird's and Yaffil's developers have chosen this way, preferring not to provide users with such an ambiguous feature.

Type conversion through COMPUTED BY

Author:
IBSurgeon Research Labs, research@ib-aid.com

InterBase has an interesting undocumented feature. Usually when the COMPUTED BY field is declared, the following syntax is used:

```
<col_def> = col { datatype | COMPUTED [ BY] (< expr>) |
domain}
    [ DEFAULT { literal | NULL | USER} ]
    [ NOT NULL] [ <col_constraint>]
    [ COLLATE collation]
```

As you see, syntax requires specifying either type of the column (datatype), or the calculated expression (computed by). The "either/or" directive is the | symbol. Usually, type of the COMPUTED BY field is similar to the source one (which it is based on). However, it is possible to specify column type, even if it would not coincide with the source.

The following experiment can be performed. Create a table with structure as shown below:

(the PRIMARY KEY definition may be omitted, since in this case it is used only for Database Explorer tables' usability: BDE does not allow update tables, which do not have a primary key)

```
CREATE TABLE TESTCOMP(
t_data  FLOAT                    NOT NULL PRIMARY KEY,
c_int        INTEGER              computed by (t_data),
c_num     NUMERIC(15, 2) computed by (t_data),
c_char     CHAR(20)               computed by (t_data))
```

Now, in the table, try to enter a record with the following T_DATA value: 1.88, 3.2, 3.51 (this test was done in dialect 1). You would see that the values the FLOAT field stores on disk differ from what you have entered. The C_INT field contains the rounded value of T_DATA. The C_NUM field would contain either exact or rounded value of T_DATA. It depends on the parameter value of the BDE ENABLE BCD = TRUE/FALSE alias. At the same time, C_CHAR would contain more precise value of the C_DATA real number. When doing this trick, it would be helpful to view the NUMERIC(15, 2) values as strings. The thing is that real numbers' accuracy cannot be stored as integers' one, and therefore when one enters 1.88, in NUMERIC(15, 2) it would look as 1.88, though actually (as a string) will turn out to be 1.8799999952316.

Thus, we can make up several conclusions:

- real numbers' accuracy is bounded, and therefore numbers stored as real, should never be used in equality (everyone knows that)
- precision of the FLOAT fields is quite short (similar to Delphi's single). That is why it is better to use DOUBLE PRECISION instead.
- not all types are interconvertible: the NUMERIC(15, 2) field as INTEGER COMPUTED BY... will contain 0.
- inaccuracy should be taken into account when processing real numbers (rounding, aggregation, comparison, addition/subtraction and multiplication/division). Also, do not forget the bookkeeper's rule: when multiplying and dividing, multiplication should be calculated in the first place.
- it is not recommended to use real types as table primary keys. Due to inaccuracy and/or peculiarities of how client's and servers processors handle real numbers: seemingly one and the same number may lead to different results.

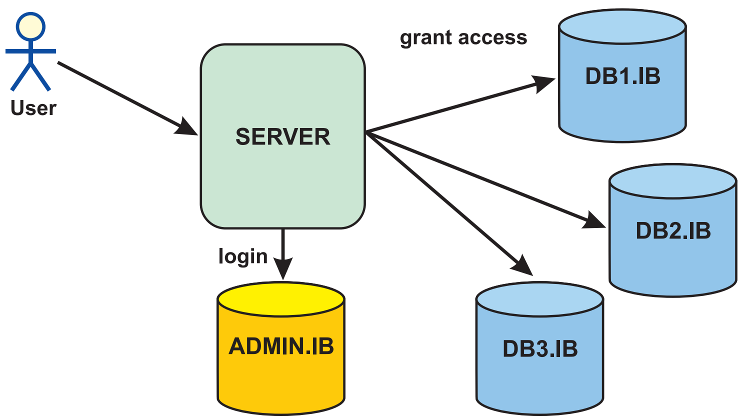
Embedded User Authentication in InterBase 7.5

Author: Dmitri Kouzmenko
kdv@ib-aid.com

One of the most important new features in InterBase 7.5 is user authentication, which is embedded in the database. Let's consider the scheme used in the previous version:

Standard scheme

In InterBase, a separate special database isc4.gdb is intended for user list storing. In InterBase 7.0 this database was renamed (admin.ib), and in addition, new ibconfig parameter was added (ADMIN_DB), which allows to specify any name for this database.



In isc4.gdb/admin.ib, there is a basic "USERS" table, which contains username, password, and other parameters. When a client connects to a database:

1. the front-end encrypts the password by DES algorithm with data loss, and then sends the username and encrypted password to the server.

2. the server encrypts the received password once again by the same lossy DES, and then calls isc4/admin in the "USERS" table, finds the necessary user, and then verifies the received password with the stored one.

3. if the passwords are equal, the user connects to the database he/she specified. And if they do not, the user is unable to connect to the database ("wrong user name or password" error reported).

As you can see, to access any database on this server, a user must be specified in isc4/admin only once. In the future, in a particular database, user access is defined by the rights he is granted.

This scheme is insufficient when used in:

• single-user applications. It becomes necessary to deploy both the database and admin.ib.

• deployed or stolen databases. Anyone can "slip" his/her own "admin.ib" with SYSDBA/masterkey to the server, and, as a result, completely control a database.

• systems, in which a user has to connect to only those databases, with which one is allowed to work.

Embedded User Authentication

In InterBase 7.5 you can either refuse using admin.ib (see below), or combine admin.ib with user control in the database. For that purpose, attributes of several system tables were extended, and new SQL-operators were added to manage this functionality (in "gsec" a "user_database" option is added for user management in such databases).

This functionality is supported only for ODS 11.2, i.e. for the databases created or restored from backup in InterBase 7.5. At that, previous versions of InterBase, for example, 7.1 and below, when attempting to connect to such database, will return two types of messages:

• product DATABASE ACCESS is not licensed

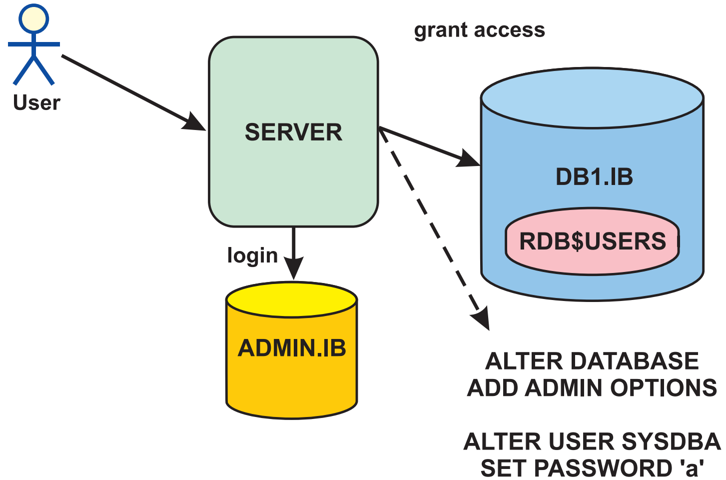
for databases, in which EUA is enabled or disabled

• internal gds software consistency check (decompression overran buffer (179), file: sqz.c line: 229)

for databases, in which EUA was never enabled

In other words, there is no other way to connect to the database, but from InterBase 7.5 specifying a required password (stored in the database) for a specific user (we do not consider the possibility of "hacking" such database, i.e. editing it in HEX-editor).

continued on page 12



Enabling EUA

There are two ways of enabling EUA in a database:

1. When creating a database, specify an extra option – WITH ADMIN OPTION – in CREATE DATABASE

2. For any ODS 11.2 database, enter the following operator

```
ALTER DATABASE ADD ADMIN OPTION
```

In any case, among ODS 11.2 database system tables there always is the RDB$USERS table. It is an equivalent to the USERS table from admin.ib (RDB$DEFAULT_ROLE, RDB$USER_ACTIVE, and RDB$USER_PRIVILEGE columns are added).

When enabling EUA, it becomes active right away, and in the RDB$USERS table the SYSDBA user standard record appears (the password is encrypted "masterkey"), and with RDB$USER_PRIVILEGE = 1. After that, when connecting to a database, the server ignores presence (or absence) of the user in admin.ib, as well as his/her password. That is to say, when EUA is enabled, one can connect to a database only if username/password combination, stored in rdb$users, is correctly specified.

EUA can be temporarily deactivated by the command

```
ALTER DATABASE SET ADMIN OPTION INACTIVE
```

and activated

```
ALTER DATABASE SET ADMIN OPTION ACTIVE
```

During deactivation, in RDB$USERS field RDB$USER_ACTIVE is set to 'N' for all user records (including SYSDBA). When activating, it is performed conversely: RDB$USER_ACTIVE is entered to 'Y' for all users. Doing that, be careful, since if some users were disabled before EUA deactivation, as soon as EUA is activated, all EUA users will be able to access the database (i.e. all EUA accounts will be enabled).

You can completely delete EUA, together with all user records by the command:

```
ALTER DATABASE DROP ADMIN OPTION
```

This will clear the RDB$USERS table, and restore functioning of the standard authentication scheme (through admin.ib).

User management

If EUA is enabled, you can manage users:

```
{ CREATE | ALTER} USER  SET

option :            PASSWORD
               [ NO] DEFAULT ROLE
                     [ NO] SYSTEM USER NAME
                     [ NO] GROUP NAME
                     [ NO] UID
                     [ NO] GID
                     [ NO] DESCRIPTION
                     [ NO] FIRST NAME
                     [ NO] MIDDLE NAME
                     [ NO] LAST NAME
                     ACTIVE
                     INACTIVE
```

Examples:

```
CREATE USER TEST SET PASSWORD 'TEST', NO LAST NAME,
DEFAULT ROLE ABC
```

As a result, a user TEST with "TEST" password will be created; the LAST_NAME column will be set NULL, the default role will be "ABC" (and rdb$user_privilege = 0, i.e. "not a database owner"). The same can be performed by the following command set:

```
CREATE USER TEST SET PASSWORD 'TEST';
ALTER USER TEST SET  NO LAST NAME, DEFAULT ROLE ABC;
```

Draw attention to the fact that one can "enable" and "disable" users by the alter user xxx set inactive/active command. There is no such possibility in the standard admin.ib.

Authentication order

It is important to comment how exactly connections are performed in case EUA is active in a database:

- The server opens a specific database.

1. EUA disabled – user authentication is accomplished from admin.ib

2. EUA enabled - user authentication (any user, including SYSDBA), is accomplished from rdb$users of this particular database

That is, when enabling SYSDBA and changing password for SYSDBA, it will be possible to connect to this database under the "SYSDBA" name, only if the user specifies a correct password.

Attention: Admin.ib is mandatory in any case. The server, when trying to connect to a database, requires presence of admin.ib regardless of whether database EUA is enabled or not.

continued on page 13

User schemes combining

Thus, in InterBase 7.5 two schemes of user management are supported: standard and EUA. This allows building the following schemes

1. standard: all users included in admin.ib are allowed to access all databases. Access rights to a specific database are defined by grants

2. EUA: the usernames for a particular database are specified in this database only. Accordingly, only these users can connect to it.

3. standard+EUA 1: SYSDBA everywhere is the same (including password), i.e. it administrates all databases on the server. The server databases can be divided into 2 sets: the first set without EUA (public access from admin.ib), and the second set with EUA (only the users specified in this particular database are allowed to access it)

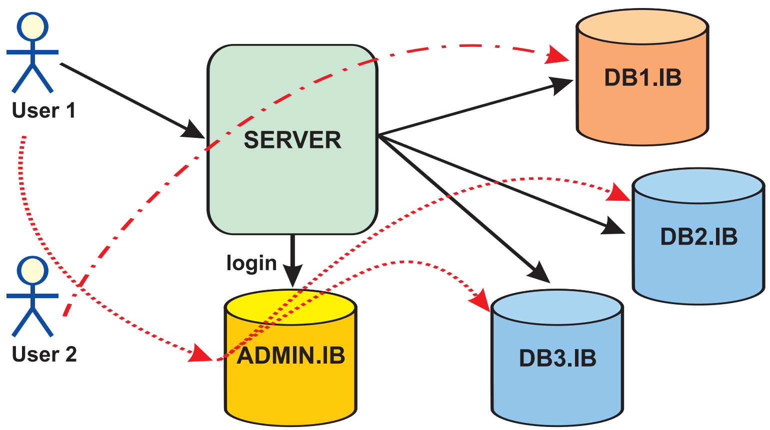
4. standard+EUA 2: All users are common for all databases (if those, for example are copied from one source), but SYSDBA requires different passwords. That is to say, one SYSDBA manages the databases, which do not have EUA, while other SYSDBA controls the databases with EUA enabled.

5. standard+EUA 3: SYSDBA usernames and passwords are different for all databases – with or without EUA.

The picture illustrates an example of how two different users connect to different databases.

User 1 can connect to databases with EUA disabled. To access DB1.IB, it is necessary to create a new user (USER1) in this database, and specify either the same, or a different password (if needed).

User 2 can connect to DB1.IB only. If this user is specified in ADMIN.IB, he/she will be able to work with the databases, in which EUA is disabled.



BACKUP/RESTORE

At the given moment, in InterBase 7.5.0.174 the following behavior is detected (there is no report about fixing that problem in IB 7.5 SP 1):

InterBase Myths № 3

Database files (gdb) must be shared with users

Never do that! InterBase – is not a file server, and works with databases independently. A client only informs the server, which database he/she wants to work with, and what queries are to be executed.

After restore, the rdb$user_privilege column of the rdb$users table has null value. Even though this is "unimportant" for SYSDBA, in cases when SYSDBA is not the database owner (the owner is, say, the "TEST" user), that particular user, as well as any other users, cannot login to such database.

The situation can be corrected if one logins to this database as SYSDBA, setting "1" value in the column instead of null for the database owner, and "0" for all other users. After this procedure, EUA's workability will be restored.

To date (16.05.2005) in Borland it is considered as bug IB 7.5.0.174.

A workaround:

disable EUA before backup (alter database set admin option inactive); after restore is performed, enable EUA (alter database set admin option active). However, to avoid change of the owner (unless it is sysdba), there should be an owner of the database with EUA in admin.ib.

Other issues

Sometimes, for different purposes, a database can be created by a user other than SYSDBA, for example, in order to use a database owner as a "backup user" (at that, all objects are created and modified on behalf of SYSDBA, and the owner cannot change them). In this case, a user who created the database, is the database owner, and thus can perform backup/restore being an owner not only of the database but also of all objects created by her/him. There are several features of applying such method when EUA is enabled.

1. Create a database not as a "SYSDBA," but as a "TEST" user. As soon as a "TEST" user is created, it becomes a database owner. At this point, of course, the TEST (with password "test," for example) user should be specified in admin.ib.

2. Enable EUA in the database.

ALTER DATABASE ADD ADMIN OPTION;

In the RDB$USERS table a record about the TEST user with "test" password appears (the password is double-encrypted, as in admin.ib), rdb$user_active = Y and rdb$user_privilege = 1

3. Add a "local" user USR

create user USR set password 'usr';

4. All this leads to an interesting situation. The "TEST" user can perform backup, but would it be a backup from admin.ib, or from the database? Let's change the TEST's password in admin.ib. Let the password be "tttt".

5. try to backup from TEST user through admin.ib

gbak -b db.ib db.ibk -v -user TEST -pass tttt

does not pass. Try a user from EUA

gbak -b db.ib db.ibk -v -user TEST -pass test

it does pass. That is, only the user specified in eua can do backup (i.e. database owner).

6. So far, it seems like one can delete the "TEST" user in admin.ib, or completely delete admin.ib. But without admin.ib the server will not connect even to the databases with active EUA. In addition, restore should be done by a user specified in admin.ib, since when restoring, it is impossible to find out whether a database has EUA or not.

gbak -c db.ibk 1.db -v user TEST -pass test

does not pass, as it was expected. The TEST user has a different name in admin.ib.

gbak -c db.ibk 1.db -v user TEST -pass tttt

restore is successfully accomplished.

However, as it was already said above, the column rdb$user_privilege = NULL. This makes impossible for any EUA user to connect to the restored database EUA (including the "TEST" user with password "test").

Connect as TEST/tttt, set "0" instead of "null" in the rdb$users column of the TEST record, and then disconnect... As a result, EUA resumes work (see above an example of temporary solution of the problem).

Conclusion

Regarding all that, we can come to several conclusions:

• EUA in this version does not "outlive" backup/restore not only for the owner, but for SYSDBA as well. SYSDBA is "both king and god" for databases and that is why rdb$user_privilege = null is unnoticeable for a developer. This becomes important when one begins to use the database in operating mode.

• When performing restore, there is a need to verify username/password. For regular databases, restoring with other usernames is performed when it is necessary to change the database owner. However, since EUA is enabled for users within the database, restore can be performed in no other way but specifying a new user at gbak -c. Actually, SYSDBA from admin.ib may be a "wrong user" differing from SYSDBA in the database with EUA, if they have different passwords.

• Probably, it would be better to create a flag in Header Page of the database, which would signify presence of EUA. In this case, users would be ignored during restore process. This is up to InterBase 7.5 developers.

• In spite of apparent "autonomy" of EUA, if there is no admin.ib, it would be possible neither to connect to the database, nor perform restoring.

Using KEEPALIVE-sockets to detect and release hung InterBase and Firebird client connections, or how to avoid the 10054/104 errors

Author: Vasiliy Ovchinnikov
ova@tkvc.ru

Introduction

In the systems within InterBase or Firebird databases, which are intended for working in either real-time or near-real-time modes, there is a problem of client connection status tracking on the server side, and of forced disconnection in case the client becomes inaccessible due to connection release. It is important to promptly release the resources busy with such phantom connections, especially when using servers with Classic architecture.

If some users connect to the server through an unstable modem connection, then the risk of disconnection becomes rather high.

For instance, a client saves a modified record set, and after UPDATE is executed (while COMMIT is not) the connection is released.

As a rule, client applications in such situations reconnect to the server, but the client (as he/she continues working with the data, after saving which one received error message due to connection fail) will be unable to save changes, since he/she will receive a message about lockout conflict ("lock conflict on update"). The previous connection, which opened the transaction (in the context of which UPDATE was executed, while COMMIT wasn't), still holds these records.

Connection failures may occur in a local network too, if the hardware (net-cards, hubs, commutators) is out of order or not adapted well, and/or due to clutter in the network. In Interbase and Firebird logs, failures of tcp connections are displayed as error 10054 in Windows and 104 in Unix; net-beui failures are displayed as 108/109 errors.

Hung connections control methods

In InterBase and Firebird, the mechanisms of DUMMY-packets or KEEPALIVE-sockets are used for tracking and disabling of such "dead" connections.

In InterBase 5.0 and higher, the mechanism of DUMMY-packets is implemented at the application layer between an InterBase/ Firebird server and a gds32/fbclient client library. It is included in ibconfig/ firebird. conf and is not examined in the present article.

Note: *As we know from previous experience, stability of the dummy-packet mechanism (the one implemented in InterBase 5.0 and repeatedly corrected in Firebird 1.5.x) strongly depends on server's and client's operating systems, tcp stack versions, and many other conditions. That is to say, effectiveness of such system in a real network tends to zero.*

KEEPALIVE-sockets are a more interesting mechanism. Implemented in InterBase 6.0 and higher, it is intended for connection failure tracking. KEEPALIVE is enabled by setting the SO_KEEPALIVE socket option at the opening. There's no need to manually set it if you use Firebird 1.5 or higher, since it is implemented in the program code of the Firebird server, both for Classic, and for Superserver.

For Interbase and Firebird versions lower than 1.5, in the variant with Classic architecture, an additional setting is necessary. This setting is described below.

In this case, the operating system TCP stack (instead of the Firebird server) becomes responsible for connection status. However, to enable this mechanism, one must adjust KEEPALIVE parameters.

KEEPALIVE description

KEEPALIVE-sockets behavior is controlled by the parameter presented in the following table.

Parameter	Description
`KEEPALIVE_ TIME`	Time interval, on expiry of which KEEPALIVE-probes start
`KEEPALIVE_INTERVAL`	Time interval between KEEPALIVE-probes
`KEEPALIVE_PROBES`	Number of KEEPALIVE-probes

The TCP stack tracks the moment when packets stop transmit between the client and the server, by launching the KEEPALIVE timer. As soon as the timer reaches the KEEPALIVE_ TIME point, the server TCP stack would execute the first KEEPALIVE probe. Probe is an empty packet with ACK flag sent to a user. If everything is alright on the client side, then the TCP stack on client side sends a response packet with ACK flag, and the server TCP stack resets the KEEPALIVE timer as soon as it receives a response.

If the client does not response to the probe, the probes from the server continue to be sent. Their quantity equals to the KEEPALIVE_ PROBES value; they are executed at the KEEPALIVE_ INTERVAL time interval. If the client does not respond to the last probe, then after another KEEPALIVE_ INTERVAL time expires, the operating system TCP stack closes the connection, and the server (in this case, instance of InterBase or Firebird server) releases all resources busy with provision of this connection.

Thus, a failed client connection will be closed after the following time interval: KEEPALIVE_ TIME+ (KEEPALIVE_ PROBES+1)* KEEPALIVE_ INTERVAL.

By default, the parameters values are rather big, and this makes use of them ineffective. For example, the default value of KEEPALIVE_TIME parameter is "2 hours," both in Linux and in Windows. Actually, 1-2 minutes would be enough to make a decision about forced disconnection of an inaccessible client. On the other hand, KEEPALIVE default settings sometimes cause forced disconnections in Windows networks, which are stay inactive during these 2 hours (of course, one may cast doubt on necessity of such connections in the applications, but this is a different matter).

Below adjustment of these parameters for Windows and Linux operating systems is described.

Setting KEEPAILVE in Linux

KEEPALIVE parameters in Linux can be changed either by file system direct editing / proc, or by calling sysctl.

For the first case, the following lines should be edited:

```
/proc/sys/net/ipv4/tcp_keepalive_time
/proc/sys/net/ipv4/tcp_keepalive_intvl
/proc/sys/net/ipv4/tcp_keepalive_probes
```

For the second case, the following commands should be executed:

```
sysctl -w net.ipv4.tcp_keepalive_time=value
sysctl -w net.ipv4.tcp_keepalive_intvl=value
sysctl -w net.ipv4.tcp_keepalive_probes=value
```

Time value is expressed in seconds.

For automatic setting of these parameters in case of server restarting, add the following lines to / etc/ sysctl. conf:

```
net.ipv4.tcp_keepalive_intvl = value
net.ipv4.tcp_keepalive_time = value
net.ipv4.tcp_keepalive_probes = value
```

Substitute the < value> word with necessary values.

If you use version of Firebird Classic lower than 1.5, then in /etc/xinet.d/firebird the following should be added:

FLAGS=REUSE KEEPALIVE

Adjusting KEEPALIVE in Windows 95/98/ME

Register branch

HKEY_LOCAL_MACHINE\ System\ CurrentControlSet\ Services\ VxD\ MSTCP

Everything about adjustment of TCP can be found here:

http://support.microsoft.com/default.aspx?scid=kb;en-us;158474

Parameters:

• **KeepAliveTime = milliseconds**

Type: DWORD

For Windows 98, type STRING.

Defines connection inactivity time in milliseconds. When it expires, KEEPALIVE-probes start executing. Default value is 2 hours (7200000).

• **KeepAliveInterval** = 32-digit value

Type: DWORD

For Windows 98, STRING type.

Defines time between KEEPALIVE-probes (in milliseconds). As soon as the specified **KeepAliveTime** interval expires, after each **KeepAliveInterval** time (in milliseconds) KEEPALIVE-probes are sent with maximum number of MaxDataRetries. If no response comes, the connection closes. Default value is 1 second (1000).

• **MaxDataRetries** = 32-digit value

Type: STRING

Defines maximum number of KEEPALIVE-probes. Default value is 5.

Setting KEEPALIVE in Windows 2000/NT/XP

Register branch

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters\.

Everything about TCP adjustment:

2000/ NT: http://support.microsoft.com/kb/120642

XP: http://support.microsoft.com/kb/314053

The **MaxDataRetries** parameter is substituted by

TCPMaxDataRetransmissions.

All other parameters have the same names as in Windows 9x

Setting KEEPALIVE in Windows (for clients)

This setting is optional, but it possibly will reduce number of messages about connection failure if one uses unreliable communications channels. Insert to the register branch

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters

parameter **DisableDHCPMediaSense=1**. See a description of this parameter here:

http://support.microsoft.com/?scid =kb%3Bru%3B239924&x=13&y=14

Example

Let's consider adjustment of Firebird SQL Server 1.5.2 CS under Linux OS.

• Make sure that the DUMMY-packets mechanism is disabled in firebird.conf (the parameter is commented-out)

....

#DummyPacketsInterval=0

....

• Make sure there is the /etc/xinet.d/firebird configuration file

We kept everything unchanged, as it was registered during installation. Nothing needs to be added.

• Change the TCP stack parameters

```
sysctl -w net.ipv4.tcp_keepalive_time = 15
sysctl -w net.ipv4.tcp_keepalive_intvl = 10
sysctl -w net.ipv4.tcp_keepalive_probes = 5
```

• Connect to any database on the server from any network client.

• Check traffic on the server using any packet filter.

If parameters specified as /proc/sys/net/tcp_keepalive_*, within 15 seconds after everything stops in the channel, the server creates a probe. If the client is "alive," the server receives a response packet. 15 seconds after that, checking repeats, and so on.

• If a client is physically turned off (either the multiplexer or the modem unexpectedly turns off – anything is possible), then the server does not receive a response, and the server begins to send probes with 10 seconds interval. If the client does not respond to the fifth probe, then 10 seconds after that, the server process discharges, and releases resources and blockings lockouts. If the client gives any signals and responses at least to the fifth probe (if worst comes to worst), then, after another 15-seconds time-out, the server will begin send probes. And so on.

Guidelines

In conclusion, we would like to give you some advice about how KEEPALIVE values should be selected.

Firstly, determine necessary value of KEEPALIVE_ TIME. The more the value is, the later KEEPALIVE-probes would start. If you constantly see 10054/104 errors in the log of the server, and you have to delete them manually, it is recommended to increase the KEEPALIVE_ TIME value.

Secondly, the values of the KEEPALIVE_ INTERVAL and KEEPALIVE_ PROBES should meet your needs concerning before-the-fact release of already hung connections. If your users connect to the server through unreliable channels, then you probably would want to increase number of probes and the interval between them, in order to give the user a chance to detect the failure and reconnect to the server. In case clients use a DSL connection to the Internet, or access a SQL-server through a local network, it is possible to decrease the interval between KEEPALIVE-probes.

General recommendations: if you for no particular reason receive from the clients many error messages, concerning results saving, due to lockout conflict (i.e. there are no concurrent connections working with the same data), then you need to increase system's reaction to the hung connections release. Practically, the KEEPALIVE_ TIME value may be above or equal 1 min. You should yourself estimate the time the longest transaction executes, so that traffic would not be overloaded by KEEPALIVE-checks of normally working connections, which launched long transactions. The KEEPALIVE_ INTERVAL value is above or equal 10 seconds, and the KEEPALIVE_ PROBES value is above or equal 5 checks. When many users work simultaneously, remember that if you perform checking too frequently, it may considerably increase network traffic.

Also remember that in case your users actively change common data, lockout errors will occur as a result of optimum situation. In this case, you would need a correct lockout error handling in the client applications. At the same time, the application should be able to minimize occurrence of such errors.

Examples of default configuration

Finally, here are some more examples of default configurations. Downtime is the time, within which users will be unable to update data, (which by that moment were updated by the transaction opened by the hung connection). Total time is the time, on the expiry of which the hung connection will be closed.

• Clients use modem connections; most of transactions in the system are short; downtime is limited by 3 minutes.

InterBase Myths № 4

If I see the word NATURAL in the query, it is a bad one!

Nothing terrible.

Some data retrieval (depending on query) is better by natural table scan, than by an index scan.

It's better to check, how long time ago you refreshed statistics for indices in your database. See documentation about SET STATISTICS INDEX <index_name> statement.

```
KEEPALIVE_TIME        1 minutes
KEEPALIVE_PROBES      3
KEEPALIVE_INTERVAL    30 seconds
TOTAL  3 minutes
```

• Clients use LAN connection; most of transactions in the system are short; downtime is limited by 2 minutes.

```
KEEPALIVE_TIME        30 sec
KEEPALIVE_PROBES      5
KEEPALIVE_INTERVAL    10 sec
TOTAL  90 seconds
```

• Clients use any connections; downtime is not regulated.

```
KEEPALIVE_TIME        15 minutes
KEEPALIVE_PROBES      4
KEEPALIVE_INTERVAL    1 minutes
TOTAL  20 minutes
```

• Clients use any connections, continuous transactions are possible in the system, and downtime limit is 15 minutes.

```
KEEPALIVE_TIME12 minutes
KEEPALIVE_PROBES      7
KEEPALIVE_INTERVAL    15 sec
TOTAL  14 minutes
```

We hope that the examples we have shown would be enough for correct adjustment of TCP stack KEEPALIVE mechanism.

Working with temporary tables in InterBase 7.5

Author: Dmitri Kouzmenko
kdv@ib-aid.com

In InterBase 7.5, a new capability of working with temporary tables was added. Unlike system temporary tables (tmp$), these tables may be created and used during applications' work. To this very day, developers had to store temporary data in ordinary tables, and that required constant table content tracking, as well as specific organization of work with data.

Surely, most often temporary tables were necessary to those developers, who had been working with MS SQL before they started to use InterBase/Firebird.

Let's consider what temporary tables in InterBase 7.5 really are.

Metadata

On the low system level temporary tables are implemented as permanent tables. That is, when you create these tables, information about them is stored in the RDB$RELATIONS system table; pointer page and other system pages are distributed for them as for regular tables. Moreover, these tables not only will be stored in the database constantly, but also will "outlive" backup/restore (as distinct from any other attempts to extend or change the structure of the rdb$ system tables).

Syntax of creation of temporary tables is as following:

```
CREATE GLOBAL TEMPORARY TABLE <table> (
        table-element-comma-list )
[ON COMMIT {  PRESERVE | DELETE }  ROWS]
```

As you see, temporary tables differ from standard ones by the global temporary phrase. Besides, on commit is added. In IB 7.5, in the RDB$RELATIONS system table, there is RDB$RELATION_TYPE column. It contains one of the following values:

RDB$RELATION_TYPE	Description
PERSISTENT	Standard tables (custom or system), in which records are deleted only by delete+commit.
GLOBAL TEMPORARY	Temporary system tables, which display server status, connection to databases, executed queries, and so on. (TMP$DATABASE, etc)
GLOBAL TEMPORARY DELETE	Temporary tables, for which ON COMMIT DELETE ROWS is specified, i.e. the records, which will be unconditionally deleted on the commit.
GLOBAL TEMPORARY PRESERVE	Temporary tables, for which ON COMMIT PRESERVE ROWS is specified, i.e. the records, which will be unconditionally deleted on disconnection.

It is not recommended to modify this column manually; this will not result in anything good. That is to say, it is impossible to turn a regular table into a temporary and vice versa.

For a time of transaction

GLOBAL TEMPORARY DELETE **stores records only until any commit is performed** (not only in the transaction, which created them, but also of **any other transaction within this connect**). Such behavior resembles a bug, since committing of competitive transactions is not supposed to flush record view. The temporary system tables work in exactly the same way, i.e. they display updated information as soon as any concurrent transaction executes a commit. At the same time, the records created in the table are invisible to all but the current transaction. A rollback in this case is equivalent to a commit, although it is clear that rollback would also cancel all changes made in the regular tables. In the case of commit, the transaction changes will be committed, while the records in temporary tables would "disappear".

Let's create such a table, and try to work with it.

```
CREATE GLOBAL TEMPORARY table TMPTRANS (
    ID int not null,
    NAME varchar(20),
    constraint  PK_TMPTRANS primary key (id) )
  ON COMMIT DELETE ROWS
```

Now you may create a procedure, which would fill in the table with some data

```
CREATE PROCEDURE XTRANS
AS
DECLARE variable I INT;
BEGIN
  I = 0;
  WHILE (:I < 10000) DO
  BEGIN
    INSERT INTO detail VALUES (:I, 'asdfasdfasdfasdfasdf');
    I = :I+1;
  END
END
```

You may insert as many records as you need: if all you want is to check the work, 100-10K would be enough. If you want to test speed, it is recommended to begin either from 100K records or million records (for example, on my computer this procedure loads 1 million records to a database with 4K within approximately 47 seconds).

Be careful, do not commit after inserting records, otherwise the records will be lost. Performing **select * from tmptran** allows you to view the records. After commit is performed, query iteration will return an empty table.

TIP:

In case you perform these operations using a tool with automatic transaction control (such as IBExpert), you would not see any temporary records, since IBExpert executing any operator in SQLEditor, performs start/commit of 3-4 another (hidden) transactions, commit of which causes loss of record view in the *on commit delete* table.

At this moment, one may ask a question: where actually are these records? The answer is: despite of "temporariness" of the records, **the temporary table records are stored in the same way as in regular tables, i.e. on a disk.** At that, after the records are inserted and commit is executed, if one gathers statistics (for example, with the help of IBAnalyst) it would look almost like the following:

Table	Records	RecLength	VerLen	Versions	Max Vers	Data Pages	Slots	Avg fill%
TMPTRANS	1000000	31.00	0.00	0	0	51725	51725	68

For a time of connection

GLOBAL TEMPORARY PRESERVE tables store records until current connection (during which they were added) is released. At that, they can be displayed only within the period of this connection.

Let's create a table

InterBase Myths № 5

InterBase is designed for Windows, and therefore it is incompatible with Unix (Linux, Solaris, etc).

This is not true! First version of InterBase was created for Unix, and, before the Windows-version was released, there were 15 "ports" for different Unix versions (AIX, IRIX, SCO, HP-UX...). Actually, the Windows version was released 7-8 years after the first version of InterBase.

```
CREATE GLOBAL TEMPORARY table TMPCONN (
    ID int not null,
    NAME varchar(20),
    constraint PK_TMPTRANS primary key (id) )
  ON COMMIT PRESERVE ROWS
```

Create a record in this table (it also can be done in IBExpert)

```
INSERT INTO TMPCONN VALUES (1, 'a')
```

Perform commit. Now, within this connect, the record will be visible from different transactions. If another instance of IBExpert (or any other tool) runs, and you execute the same insert operator, it would be executed with no PK or UNUQIE key violation error.

As soon as you close the current connection and open a new one, the entered data will be lost.

Connections between temporary tables

It is quite interesting that you can create Foreign Keys between temporary tables, but this cannot be done between a temporary table and a constant one. However, when creating FK one should take into consideration the record view area in both tables. For example, you create two tables:

```
MASTER, on commit delete
DETAIL, on commit preserve
```

and create FK from DETAIL to MASTER. As a result, (actually, InterBase would not allow to create such FK) after creation of records in *master* and *detail*, the first commit would delete all records in master, and that causes presence of records with missing connections in DETAIL (in fact, such type of connection as *commit preserve -> on commit delete* is not permitted, though you can perform the opposite).

To do that (and to change "temporariness" type of the records), the ALTER TABLE operator has the following extension:

```
ALTER TABLE <table> ON COMMIT { PRESERVE | DELETE} ROWS
 { RESTRICT| CASCADE}
```

This operator changes table's type (preserve/delete) and can also perform cascading correction of type of the tables bound by FK, in order to prevent the situations with mismatch of records' lifetime in master and detail. The RESTRICT directive will inform about error, if other temporary tables refer to this table.

Temporary tables of all types cannot use FK pointing to constant tables.

Garbage collecting

As already mentioned, in spite of "temporariness" of table content, the **on commit delete** and **on commit preserve** records, are nevertheless stored on the disk, as in ordinary tables. Therefore, the server sometime must remove them (as garbage). This happens when the following events occur:

Table type	When garbage is collected
ON COMMIT DELETE	At first "exclusive" connection to the database
ON COMMIT PRESERVE	When canceling the connection created the record

An example of the procedure, which automatically fills a temporary table with records, is given on purpose. Tests were held using 1 million records. For the tests, 2 IBExpert instances were launched and one IB_SQL was used. Without going into details of the test, we will list its results and conclusions

• For ON COMMIT DELETE tables, garbage is collected during first exclusive connection to the database. Assume we have 10 working applications, which fill in temporary tables. To delete records in all temporary tables, all 10 applications should disconnect, and at least one should connect. Right at that moment, garbage collecting in the ON COMMIT DELETE tables begins. All connections, which attempt to connect to the server before garbage collecting is finished, will "hung".

Resume

• Working with ON COMMIT DELETE temporary folders may lead to fast grow of the database during a day, since it is very seldom that during this period of time all users disconnect from the database

• The more garbage is collected in the ON COMMIT DELETE temporary tables, the longer will be the delay between the first connection and the working. It takes approximately 25 seconds for server to delete 1 million temporary records, and ~120 seconds to delete 3 million temporary records.

• For the ON COMMIT PRESERVE tables, garbage is collected when disconnecting the connection, which created these records.

Resume

• The more records a connection creates in temporary tables, the longer the application would "hang" when disconnection is performed. Deletion of 1 million temporary records, as well as in the previous case, takes ~25-35 seconds.

Summary

Temporary tables InterBase 7.5 – are very useful for applications, which form complex reports and execute intermediate calculations on the server. However, due to strange behavior of ON COMMIT DELETE, it becomes possible to use transaction context temporary tables only in the applications, which works with only one transaction at a time. Or they can work with several transactions, on condition that a commit of competitive transaction is forbidden until the transaction (which works with the temporary table) performs a commit.

Furthermore, use of ON COMMIT DELETE tables causes collecting of garbage records during multiuser work (since database's size increases), and collects garbage on first connection to the database. This can cause an undesirable delay in the beginning of the users' work.

ON COMMIT PRESERVE is a more favorable way, though the process of disconnection of applications would be more time-consuming (of course, unless these applications created records in temporary tables). In order to avoid users' complaints, you will probably need to specially handle application disconnection, and to display a message asking to wait some time.

p.s. during the temporary tables test, a spontaneous processor loading by the IB7.5.0.28 server was observed (though the applications were inactive). At that, the loading appeared in certain order of transactions' starting and finishing, while they did not contain the executed operator. The reason of this effect is currently being ascertained (with InterBase 7.5 SP1 also).

Working with UNICODE in InterBase/Firebird

Author: Dmitri Kovalenko, LCPI
dima@lcpi.lipetsk.ru

What is UNICODE_FSS?

It is an InterBase codepage (often called UTF-8), which displays double-byte and four-byte UNICODE characters (UCS-2 и UCS-4, respectively) in character strings from 1 to 6 bytes. What is it intended for?

- It provides transport for UNICODE texts based on regular ASCII text.
- Data packing. The characters with codes less than 128 are, as usual, represented as one byte

Interconversion of UTF-8 and UCS-2 (UCS-4) files is based on use of the following table:

Table 1 Converting UCS-2 to UTF-8

Bits	Hex Min	Hex Max	UTF-8 Binary Encoding
7	00000000	0000007F	0xxxxxxx
11	00000080	000007FF	110xxxxx 10xxxxxx
16	00000800	0000FFFF	1110xxxx 10xxxxxx 10xxxxxx
21	00010000	001FFFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
26	00200000	03FFFFFF	111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
31	04000000	7FFFFFFF	1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

UNICODE (the text format one usually deals with, when programming for Windows) uses double-byte character set (UCS-2). As evident from the table above, 1-3 bytes would be enough for presentation of the UNICODE characters with a code from 0x0000 to 0xFFFF in UTF-8.

InterBase proceeds from the fact that UNICODE characters within the range [0x0000,0xFFFF] will be stored in UTF-8. Therefore, when specifying size for storing text data with UNICODE_FSS codepage, the number of characters is multiplied by 3 bytes. In addition, all current versions of InterBase, when working with CHAR/VARCHAR data in data presentation, controls number of bytes, not of characters. That is why, for example, one can enter up to 9 single-byte characters to the CHAR(3) column with UNICODE_FSS codepage.

Example

```
CREATE TABLE TESTUTF(TESTFIELD CHAR(3));
INSERT INTO TESTUTF (TESTFIELD) VALUES('123456789'); -- NO EXCEPTION!
```

The main disadvantage of working with UTF-8 is the impossibility to detect the number of symbols in a string without viewing it.

That is why, for better performance, it is recommended to recode text data in UTF-8 format to a codepage with characters of fixed size (such as UCS-2).

Creation of database using UNICODE_FSS

Nothing out of the way, everything is as usual:

```
CREATE DATABASE ... DEFAULT CHARACTER SET UNICODE_FSS.
```

Connecting to a database with a UNICODE_FSS codepage

As is well known, when creating a database, one can use one codepage, and when connecting to a database, can use a different codepage can be used. It's partly true since a server when interacting with a client, tries to recode text data to a coding the client wishes to use.

However, this mechanism has constraints and exceptions, which are described below. That is why we recommend not to experiment with it. When connecting to a database, specify the codepage, which was used during database creation.

In our case, it is UNICODE_FSS. This means that we want to get text data in UTF-8 format from the database, and the text will pass the text in the same format.

Working with database

Exchange with server uses the following data categories:

- General text fields with CHAR and VARCHAR types
- BLOB text fields
- Arrays
- SQL query text.

Text fields

When reading text fields, the server requires providing a buffer: (max character size)*(number of characters). This is the value, which will be put in XSQLVAR.sqllen after data access query text is prepared.

A user does not have to worry about calculation of size needed for text field data in the clipboard. We recommend using the value specified in the definition of the field. However, do not forget that, for the VARCHAR columns (SQL_VARYING type), one should add 2 bytes to the specified value, in order to reserve some space for column length indicator. The returned value will be expressed in bytes.

The thing is that in win1252 all symbols are single-byte, and buffer for such column will be required reasoning from 1 byte for 1 character. When recoding to UTF-8, characters with a code more than 127 will become at least double-byte, and this may result in overflow.

So please consider it, if you want to use connections with several codepages in the context of a single database.

BLOB text fields

As is well known, the link to data is stored in the InterBase record, which contains a BLOB-field (subtype independent). The data are stored separately and handled by InterBase API functions, designed specially for working with BLOB.

TIP:

By the way, one of the undocumented features of InterBase is that for text fields in XSQLVAR.subtype the number of the field codepage is specified in the lower byte, while the collate number is specified in the upper byte.

When recording text fields, you convert a text to UTF-8, and then handle it as an ordinary record. Buffer size and line length parameters are expressed in bytes.

If the field codepage differs from the codepage of connection to database, then the server performs conversion of incoming and outgoing data.

However, for all that, buffer size will be computed according to the column codepage. Thus, if the codepage of the win1252 field and UNICODE_FSS connection is used, and no other additional operations are launched, then you will probably receive the «Cannot transliterate characters between character sets» error message.

That is to say, BLOB-fields data are read and written with separate calls of API. That is why the rules of working with text information in BLOB differ from the ones of working with CHAR/VARCHAR.

The basic rule is: take care of yourself. By default, the server does not interpret contents of BLOB-fields, and treats them as ordinary binary information. Thus, the client becomes responsible for data recoding.

Usually it is enough to use UNICODE_FSS coding on the client, and not use UNICODE_FSS and one-byte coding simultaneously, since conversion to one-byte coding can cause data loss.

InterBase Myths № 6

Compiled procedures store query plans.

Not a bit (unless the query plan is explicit)!

This myth is based on the fact that the procedure after first call (that is the moment when query plans, written in the procedure, are computed) remains in the metadata cache until all clients, who called this procedure, disconnect. In this case, actually, while the procedure is in memory, query plans do not change, even if statistics of the indexes used by the plans change. Read about this in InterBase documentation - DataDef.pdf, Chapter 9, section "Altering and dropping procedures in use".

Arrays

UNICODE support in text arrays is similar to the one for text fields.

As in the text fields case, the server, when working with text arrays, operates on the byte-level, not on the level of characters. Therefore, the number of characters in a string written in an array cell, may exceed the length specified when the text array column had been created.

In exactly the same way, the server supports recoding of input and output arrays data, taking into account the connection codepage.

SQL query text

Strange as it may seem at first sight, SQL query text is also must use a codepage for connection to a database. The point is that a SQL query text is one of the methods of parameter values explicit transfer.

There are no severe limitations, except for general length of an SQL query, which is 64K. When converting a query containing national coding characters to UTF-8, the resulting text may be larger than the source one, and thus it would exceed 64K limit.

Access components for working with UNICODE_FSS

Generally speaking, support of a specific codepage means that the access to database component is able to ensure client's work with other codepage. To do that, it is enough to guarantee conversion of text data (at that, UCS-2, an intermediate format is used). Conversion of text columns, arrays and SQL-queries is not a problem at all, but BLOB fields conversion is quite a laborious task, especially when accessing the BLOB field data through a stream mechanism.

Theoretically, client application, of course, should not depend on the codepages' differences. Therefore, access components must block all possible ways of text data transfer, and provide necessary information recoding.

Hyperthreading, SMP and InterBase, Firebird, Yaffil

Author: Dmitri Kouzmenko
kdv@ib-aid.com

When one uses either Firebird/Yaffil Classic or InterBase 7.1/7.5, the following questions are often asked: "what server to choose, a multiprocessor, or a single-processor one?" and "should Hyper-Threading be enabled on the server or not?"

It's quite possible that if the server is purchased from the hardware provider, such questions do not arise, since in such cases basic configuration is used (for example, a dual-processor server with HT).

However, different tests and comments witness to the fact that it makes sense to use (keep enabled) the hyperthreading technology only on single-processor workstations (!), and not on servers.

There are many reasons why Hyperthreading should be turned off on the server:

Note: *Each paragraph contains a link to a document, which exemplifies "why it is so." But this does not mean that the conclusion in a particular paragraph is drawn from that document. You, of course, will be able to find on the Internet corroboration of the described facts and our experience. We followed our own experience and tests results, and we have analyzed some reports (tests) of other people, as well as many different documents concerning HyperThreading.*

1. Database-application (Firebird, Interbase, Yaffil) is an application, which actively uses both the processor and disk memory. At the same time, the percentage of processor and disk charge may vary.

Total charge of several real (as well as virtual) processors can occur only on "calculating" applications.

2. Enabling HT on a double-processor computer leads to appearance of four processors (which are 2 physical and 2 virtual). Thus, when all processors are used, the processor bus becomes more loaded. As a result, instead of productivity increase, the system performance would slowdown by 10-15%.

3. According to different reports, enabling HT slightly increases productivity, decreases it, or does not affect productivity at all. Therefore, the result can be seen only when analyzing a particular combination of motherboard, memory, and processors.

[conferences на newsgroups]

4. As Intel and Microsoft claim, Windows 2003 is the only operating system "certified" for HyperThreading. Therefore, if one uses either Windows NT or Windows 2000, it is most likely that if HT is enabled, productivity would be the same or would decrease. The same is true for Linux, i.e. HT should be enabled using appropriate versions of this OS.

http://www.intel.com/support/platform/ht/os.htm?iid=ipp_htm+os&

Note: *InterBase 7.1 tests on Windows 2000 and Windows2003 with enabled HyperThreading have shown that Win 2000's productivity becomes worse. At the same time, it is similar for both OS', if HT is disabled. That is why the ibconfig ENABLE_HYPERTHREADING parameter for IB 7.1 is disabled by default.*

5. Generally, there are applications, which may work incorrectly (fail) on multiprocessor computers if Hyper-Threading is enabled.

(search Google for bsod+hyperthreading)

6. The HyperThreading technology is designed for increasing of multi-thread applications' productivity. Therefore, execution speed of two processes on two virtual processors would be slower than it is on two physical processors. If in addition to a database, some applications are active on the server, then enabling HT would cause decrease of general productivity (this does not concern single-processor systems, in which a slightly increase of productivity may be).

http://www.intel.com/technology/hyperthread/

Supplement

While we hope you understood that HyperThreading must be turned off on server, it is still not clear whether it is better to use a single-processor server, or a multiprocessor one. The answer to that question depends on architecture of the server and operating system:

In the "2 and more processors " column:

"No" – not recommended or it makes sense to tie a server process to a specific processor (for example, for Windows through the ibaffinity utility or through the CPU_AFFINITY parameter in ibconfig/firebird.conf).

"Yes" – this server variant with this operating system will use all processors. In IB 7.x, it is required to purchase processor license for each additional physical processor.

For clarity sake, "yes" variants in the table are typed in bold.

Server	Server type	Operating system	2 and more processors
InterBase 6.0	SuperServer	Windows	No
		Linux	No
		Solaris-SPARC	No
	Classic	Linux	**Yes**
InterBase 6.5	SuperServer	Windows	No
		Linux	No
		Solaris-SPARC	No
InterBase 7.x	SuperServer	Windows	**Yes**
		Linux	**Yes**
		Solaris-SPARC	**Yes**
Firebird 1.0	SuperServer	Windows	No
		Linux	No
		Solaris-x86	No
		HP-UX	No
	Classic	Linux	**Yes**
		Solaris-x86	**Yes**
		Solaris-SPARC	**Yes**
		FreeBSD	**Yes**
		MacOS/X	**Yes**
		HP-UX	**Yes**
Firebird 1.5	SuperServer	Windows	No
		Linux	No
	Classic	Windows	**Yes**
		Linux	**Yes**
		FreeBSD	**Yes**
		MacOS/X	**Yes**
		Sinixz	**Yes**
		Solaris-x86	**Yes**
Yaffil	SuperServer	Windows	No
	Classic	Windows	**Yes**