

Firebird / Interbase Transaction Simulator

(Felix John COLIBRI)

- **abstract** : the transaction simulator uses either scripts or random commands to simulate the Firebird database versioning system and the result of the simulator are checked with Firebird's outputs. Read committed and Snapshot transactions are presented. Some remarks about long transactions and Interbase / Firebird slowdown caused by transactions
 - **key words** : Interbase, Firebird, Transaction, Versioning database systems, Multi Generational Architecture, Read Committed, Snapshot, long transactions, transaction lifetime, OAT, OST, OIT
 - **software used** : Windows XP Home, Delphi 6, Firebird 2.5, lbx
 - **hardware used** : Pentium 2.800Mhz, 512 M memory, 140 G hard disc
 - **scope** : Delphi 1 to 2006, Turbo Delphi for Windows, Kylix
Delphi 5, 6, 7, 8 Delphi 2005, 2006, Delphi 2007 to Delphi 2009, Delphi Xe2 to Delphi Xe4
 - **level** : Delphi developer, Interbase / Firebird Developer
 - **plan** :
 - [Firebird Transaction Simulator](#)
 - [Database Versioning Systems](#)
 - [The Delphi Transaction simulator](#)
 - [Firebird Transactions](#)
 - [Interbase / Firebird transaction optimization](#)
 - [Download the Delphi Source Code](#)
-

1 - Firebird Transaction Simulator

We all know that transactions are at the heart of Firebird's architecture. So understanding how they work could be helpful for selecting the appropriate techniques to keep this engine work optimally.

The best way to understand what's under the hood seemed to simulate how this transaction / versioning system works. And to be reasonably sure that our program correctly represents how Firebird works, we also compared the results to Firebird's own outputs.

Of course we only represented a tiny fraction of the full Firebird engine. So to put the topic in perspective we also added some comments describing all the transaction options available and added some comments about how transactions could impact Firebird's responsiveness.

2 - Database Versioning Systems

2.1 - Basic operations

Our simple database uses an bank account table, with two fields in each row

- the name of the holder of the account, which is the primary key
- the amount deposited on the account

We want to perform the basic CRUD operations on this table (**C**reate a row, **R**ead, **U**update and **D**delete it).

To guarantee ACID (**A**tomicity, **C**onsistency, **I**solation, **D**urability) properties, we are using transactions :

- each CRUD operation happens in the context of a transaction
- the operations on a transaction are
 - start a new transaction
 - end this transaction by calling
 - `commit` : all row operations since this transaction started are considered valid
 - `rollback` : all operations are invalid, as if they never happened
- a transaction can therefor be in 4 states
 - `active` : has been started, has not yet been committed or rolled back
 - `committed` : `commit` has been called
 - `rolled back` : `rollback` has been called
 - for Interbase / Firebird there is a fourth possible state, which is limbo. This only happens when some program opens 2 database. We did not investigate this case.

2.2 - Notation

To present the different operations, we use the following notation:

- `c r u d` (lowercase) are used for the CRUD row operations (create, read, update, delete)
- `S C R` (uppercase) denote the transactions operation (START, COMMIT, ROLLBACK)

Here is an example (script 1):

```
START T1
c T1 A 800
r T1 A
COMM T1
```

which can be read as

- START transaction T1
- in the context of T1, create a row for customer A who deposits \$ 800
- in the context of T1, read the amount deposited by customer A
- COMMIT transaction T1

We call each line an "action". For CRUD an action is not an SQL request but is the low level implementation of SQL requests ("create" implements INSERT, "read" SELECT, "update" UPDATE and "delete" DELETE).

And the sequence of actions constitutes a script.

2.3 - Transaction Isolation

We can use as many transactions as we want (script 2):

```
START T1
c T1 A 800
r T1 A
COMM T1

    START T2
    c T2 B 800
    r T2 B
    COMM T2
```

Our transactions have a transaction id. Those ids are increasing sequentially. T1 is "older" than T2 (T1 started before T2).

Also Many transactions can be active at the same time (script 3):

```
START T1
c T1 A 800
    START T2
    c T2 B 800
r T1 A
    r T2 B
    COMM T2
COMM T1
```

In the previous example, the two transactions operate on 2 different rows (key A and key B). But active transactions might also handle the same row. This could lead to some "anomalies". For instance (script 4),

```
START T1
c T1 A 800
  START T2
    r T2 A
```

If T1 rolls back, the "A 800" value is removed, and T2 has read a value which has been later removed.

To avoid this, we will only allow a transaction to read the values which have been committed. So the previous operation would cause an error. This sequence would allow T2 to read "A 800" (script_5):

```
START T1
c T1 A 800
  START T2
    COMM T1
      r T2 A
```

2.4 - Record Modification Lock

The second rule is to forbid the modification of the same row by two active transaction (script 6) :

```
START T1
c T1 A 800
COMM T1
  START T2
    u T2 A 900
      START T3
        u T3 A 1000
```

When those transactions commit, what should the account amount be: it could be 900 or 1000 depending on the order of the commits or introducing some other rules.

So in our case, we will forbid this situation. The first transaction that modifies (update or delete) a row places a lock on this row, and any attempt by another active transaction to modify this row will return an error.

This lock will be removed when the transaction which placed the lock commits or rolls back.

2.5 - Row Versions

Since the modifications can be rolled back, we must save the values which were modified by the transactions which rolls back.

Basically two techniques exist

- save the values before any modification and restore those values when the transaction is rolled back
- save the initial value of a key as well as any later modifications, remembering which transaction saved the value

Interbase / Firebird use the second solution wich is a versioning system.

To implement a basic versioning system, we will use

- the list of all transactions, each transaction having
 - a unique number
 - a state : active, committed, rolled back
- the list of all rows of our account table, each row having
 - a number (to allow chaining the successive versions of the same key)
 - the key
 - the amount (or any other fields)
 - a link to the previous version of this key
 - the transaction is which created this row version
 - a "deleted" flag

Here is a dump of operations on a single key (script 7):

```
actions :
START T1
c T1 A 800
u T1 A 900
u T1 A 1000

transaction list:
T1 active

row list :
101 A 800 (T1 active)
102 A 900 (T1 active) x [ -> 101]
103 A 1000 (T1 active) x [ -> 102]
```

which means

- we create a transaction T1, create a row "A 800", and updated this key with value 900 then 1000
- the transaction list contains only a single transaction which is active
- the table contains 3 rows
 - the first version, 101, contains the key A, amount 800, and was created by transaction T1, which is active at the time of this dump
 - the second row, version 102, shows the updated value 900, and the row is locked ("x") and the previous version of this key is row 101
 - similar comments for version 103

And here is another example displaying how read committed works (script 8):

```

01 START T1
02 c T1 A 800
03 COMM T1
04  START T2
05  u T2 A 801
06  r T2 A =801
07  START T3
08  r T3 A =800
09  COMM T2
10  r T3 A =801

T1 commit
T2 commit
T3 active

101 A 800 (T1 commit)
102 A 801 (T2 commit) [-> 101]

```

and

- action 06 read of "A" returns 801, since a transaction will see values modified by itself
- when action 08 reads "A", it does not see the value modified by T2 during action 05, since T2 is not committed
- when action 10 reads "A", it now returns the value modified by T2, since T2 is now committed

This example also shows that

- when new values are committed, the old one can be destroyed: here any active transaction (be it T3 or any later transaction) when reading "A" will always get the last committed value, "801". So the "101800" row version can be removed (garbage collected).
- with the read committed transaction mode, two successive reads by the same transaction can return two different values (because another transaction which

modified the value did commit between those reads). This is the case for the T3 07 and 09 read actions, with T2 committing in between.

And here is an illustration of the row modification lock (script 9):

```
01 START T1
02 c T1 A 800
03 COMM T1
04  START T2
05  u T2 A 801
06    START T3
07    u T3 A 802 *** lock_ver 102
08    r T3 A =800

T1 commit
T2 active
T3 active

101 A 800 (T1 commit)
102 A 801 (T2 active) x [-> 101]
```

and

- when at 07 T3 tries to update "A", there is an error: the update is not performed
- therefore at 08, the value read is 800 (the only read committed value)

2.6 - Exceptions and Failures

Is updating a locked row an exception or a failure ?

The final application can launch requests which do not succeed. For instance a SELECT with a WHERE with does not find any row. This is a failure. At the level of our simulator, the read action does not find any row, but this is not a mistake.

However updating a locked row, creating twice the same key will trigger exceptions.

In our displays, failures are indicated with a single star, "*", exceptions with 3 stars, "***".

And running with parallel Firebird execution, we do get exceptions. To avoid stopping the execution when running long scripts, either run from the Windows explorer or uncheck "stop on exception".

2.7 - Deleting rows

When we delete a row, if we rollback the transaction, the original value should become available again. Therefore deleting a row creates a new row version with a "deleted" flag.

In addition

- a row can only be deleted by one transaction at a time (what would happen if none, some or all of those transactions committed or rolled back ?). Therefore there the delete action places a lock on the row
- if the deleting transaction committs, the deleted row can be removed (garbage collected)

Here is an example (script 10):

```
01 START T1
02 c T1 A 800
03 COMM T1
04  START T2
05  d T2 A
06    START T3
07    r T3 A =800
08    d T3 A *** lock_v 102

T1 commit
T2 active
T3 active

101 A 800 (T1 commit)
102 A -del (T2 active) x [-> 101]
```

Suppose that T2 commits, and T3 tries to read "A" (script 11):

```
01 START T1
02 c T1 A 800
03 COMM T1
04  START T2
05  d T2 A
06    START T3
07    r T3 A =800
08    d T3 A *** lock_v 102
09  COMM T2
10    r T3 A *** committed_del

T1 commit
```

```
T2 commit
T3 active

101 A 800 (T1 commit)
102 A -del (T2 commit) [-> 101]
```

and

- the action 10 fails, since the row is deleted
- both version 101 and 102 can be garbage collected

Finally, a transaction cannot read a key it did delete before (script 12):

```
01 START T1
02 c T1 A 800
03 d T1 A
04 r T1 A *** own_del

T1 active

101 A 800 (T1 active)
102 A -del (T1 active) x [-> 101]
```

2.8 - Rolling back a transaction

Rolling back a transaction simply toggles the state of this transaction to "rolled back".

Here is an example with some updates (script 13):

```
01 START T1
02 c T1 A 800
03 COMM T1
04  START T2
05  u T2 A 801
06  u T2 A 802
07  ROLL T2
08  START T3
09  r T3 A =800

T1  commit
T2  r rolled
T3  active

101 A 800 (T1  commit)
102 A 801 (T2  r rolled) [-> 101]
```

```
103 A 802 (T2 r rolled) [ -> 102]
```

and with deletes (script 14)

```
01 START T1
02 c T1 A 800
03 COMM T1
04  START T2
05  u T2 A 801
06  d T2 A
07  ROLL T2
08   START T3
09   r T3 A =800

T1  commit
T2  r rolled
T3  active

101 A 800 (T1  commit)
102 A 801 (T2 r rolled) [ -> 101]
103 A -del (T2 r rolled) [ -> 102]
```

2.9 - Garbage collection

Rows which are no longer reachable by any active or future transaction can be garbage collected.

Looking at the row list, we can use the following algorithm

- for each key, start at the end of the list (the most recent version)
- loop back until a committed value is reached
- all versions of this key before the committed value can be removed

Here is an example (script 15):

```
01 START T1
02 c T1 A 800
03 COMM T1
04  START T2
05  u T2 A 801
06  COMM T2
07   START T3
08   u T3 A 802
09   START T4
10   r T4 A =801
11    START T5
12    COMM T5
```

```

T1 commit
T2 commit
T3 active
T4 active
T5 commit

101 A 800 (T1 commit)
102 A 801 (T2 commit) [ -> 101]
103 A 802 (T3 active) x [ -> 102]

```

We see that

- in the row list, for any key
 - because of the modification locking rule, there can only be at most a single row version with an active transaction (103)
 - this row version with an active transaction, if any, is always at the top (103, the last row version). If some transaction (T3) did modify a key, there can be no later transaction which modifies this same key
- in the transaction list
 - active, committed, rolled back can be in any order
 - there can be many active transactions, since even reading requires a transaction

The garbage collection could be triggered at any time (when the system is idle). However Firebird performs this garbage collection before reading a key.

This does make a lot of sense, since when we want to read the value of a key, we have to find the last row version of this key anyway. So why not at the same time visit all the versions of this key and remove all rows that will not longer be used by anybody ?

One could think that triggering garbage collection when committing (or rolling back) a transaction could also be a good idea. This however would force us to hunt for all the keys ever touched by this transaction.

Here is the same example with garbage collection turned on (script 16) :

```

01 START T1
02 c T1 A 800
03 COMM T1
04  START T2
05  u T2 A 801
06  COMM T2
07  START T3
08  u T3 A 802

```

```

09     START T4
10     -garb T1 A 101
11     r T4 A =801
12     START T5
13     COMM T5

T1 commit
T2 commit
T3 active
T4 active
T5 commit

101 A 800 (T1 commit) G
102 A 801 (T2 commit)
103 A 802 (T3 active) x [-> 102]

```

Note that

- garbage collection does remove the unnecessary row versions from disc. We should also with our memory object simulator remove those row versions from the table.

The table in the previous figure could be represented by (display 17):

```

102 A 801 (T2 commit)
103 A 802 (T3 active) x [-> 102]

```

To better show what has been garbage collected, we chose however to keep those removed row versions in our display, but marked with a "G"

When sifting down the row versions looking for the first committed version, if the first committed row is a delete, this row version and all row versions before can be garbage collected (script 18).

```

01 START T1
02 c T1 A 800
03 COMM T1
04 START T2
05 u T2 A 801
06 COMM T2
07 START T3
08 d T3 A
09 COMM T3
10 START T4
11 -garb T3 A 103
12 -garb T2 A 102
13 -garb T1 A 101
14 r T4 A * committed_del

```

```

T1 commit
T2 commit
T3 commit
T4 active

101 A 800 (T1 commit) G
102 A 801 (T2 commit) G
103 A -del (T3 commit) G

```

Note that

- garbage collection was triggered by the read action, even if the read could not find the key

If a transaction is rolled back, all row versions containing modifications by this transaction can be removed. Here is an example (script 19)

```

01 START T1
02 c T1 A 800
03 COMM T1
04  START T2
05  u T2 A 801
06  ROLL T2
07  START T3
08  -garb T2 A 102
09  r T3 A =800

T1 commit
T2 r rollback
T3 active

101 A 800 (T1 commit)
102 A 801 (T2 r commit) G

```

Since this garbage collection happens when some key is read, there still could remain some other key row versions rolled back by this transaction (script 20):

```

01 START T1
02 c T1 A 800
03 COMM T1
04  START T2
05  c T2 B 950
06  COMM T2
07  START T3
08  u T3 A 801

```

```

09  u T3 B 951
10  ROLL T3
11  START T4
12  -garb T3 A 103
13  r T4 A =800

T1  commit
T2  commit
T3  r rolled
T4  active

101 A 800 (T1  commit)
102 B 950 (T2  commit)
103 A 801 (T3 r rolled) G
104 B 951 (T3 r rolled) [-> 102]

```

and

- "104 B 951 (T3)" is still rolled back and not garbage collected

2.10 - Sweep

Garbage collection is triggered when we read a row. It is frequent that some rows are updated many times but not read for long periods. They will therefore accumulate long obsolete row versions (script 21).

```

01  START T1
02  c T1 A 800
03  u T1 A 801
04  COMM T1
05  START T2
06  c T2 B 900
07  u T2 B 901
08  COMM T2
09  START T3
10  c T3 C 1000
11  u T3 C 1001
12  COMM T3
13  START T4
14  -garb T3 C 105
15  r T4 C =1001

T1  commit
T2  commit
T3  commit
T4  active

101 A 800 (T1  commit)

```

```
102 A 801 (T1 commit) [-> 101]
103 B 900 (T2 commit)
104 B 901 (T2 commit) [-> 103]
105 C 1000 (T3 commit) G
106 C 1001 (T3 commit)
```

Here, only key "C" was read and its old committed row versions garbage collected. Row versions 101 and 103 could also be removed, but since there was no read for those keys, the stale row versions still occupy disk space.

Calling sweep will remove those row versions (script 22):

```
01 START T1
02 c T1 A 800
03 u T1 A 801
04 COMM T1
05  START T2
06  c T2 B 900
07  u T2 B 901
08  COMM T2
09  START T3
10  c T3 C 1000
11  u T3 C 1001
12  COMM T3
13  START T4
14  -garb T3 C 105
15  r T4 C =1001
16 SWEEP
17 W-garb T1 A 101
18 W-garb T2 B 103

T1 commit
T2 commit
T3 commit
T4 active

101 A 800 (T1 commit) G
102 A 801 (T1 commit)
103 B 900 (T2 commit) G
104 B 901 (T2 commit)
105 C 1000 (T3 commit) G
106 C 1001 (T3 commit)
```

The sweep also removes all rolled back row versions. There is an additional Firebird optimisation: after the row version removal, the rolled back transaction is toggled to

"committed" (script 23).

```
01 START T1
02 c T1 A 800
03 COMM T1
04  START T2
05  c T2 B 950
06  COMM T2
07  START T3
08  u T3 A 801
09  u T3 B 951
10  ROLL T3
11 SWEEP
12 W-garb T3 A 103
13 W-garb T3 B 104

T1  commit
T2  commit
T3 r commit

101 A  800 (T1  commit)
102 B  950 (T2  commit)
103 A  801 (T3 r commit)  G
104 B  951 (T3 r commit)  G
```

Note that

- the benefit of this is that it is easier to compute the threshold before which all transactions are committed. When a row version has a transaction id lower than this threshold, is it not necessary to perform a lookup in the transaction table, since we know that it is committed. There is no risk to test this on a rolled back row version, since they have been garbage collected (the rows have been removed, not the transaction).
- to highlight this optimization commit, we add a "r" marker for this transaction
- this optimization cannot be performed when doing garbage collection before reading a key, since there might be some other rolled back keys which are still not garbage collected. Remember, the reading performs garbage collection on this key's row versions only, whereas the sweep handles all keys from all tables

For Firebird, sweep is called

- manually by the database administrator
 - automatically when the unused row version count reaches some threshold
-

3 - Snapshot transactions

3.1 - Repeatable Reads

The read committed transaction mode works nicely, but, as already noted, suffers from the "inconsistent read anomaly": within a transaction, two succeeding read could return different values (cause by some other transaction performing an update which is committed between those reads).

For applications which mainly record some values and read some values back from time to time, this is quite alright.

However for statistical analysis type of applications, this is usually not acceptable.

If the banker wants to know how much has been deposited by his customer, he has to read each row and accumulate the amounts (script 24):

```
01 START T1
02 c T1 A 800
03 COMM T1
04  START T2
05  c T2 B 900
06  COMM T2
07   START T3
08   c T3 C 1000
09   COMM T3
10    START T4
11    r T4 A =800
12    r T4 B =900
13    r T4 C =1000
```

```
T1 commit
T2 commit
T3 commit
T4 active
```

```
101 A 800 (T1 commit)
102 B 900 (T2 commit)
103 C 1000 (T3 commit)
```

The total clearly is 800+ 900+ 1000.

If customer Joe transfers \$ 500 from his account "A" to his account "B", this could happen like this (script 25):

```
01 START T1
```

```

02 c T1 A 800
03 COMM T1
04  START T2
05  c T2 B 900
06  COMM T2
07  START T3
08    START T4      // joe starts
09    u T4 A 300    // joe removes $500 from A
10  c T3 C 1000
11    u T4 B 1400   // joe deposits $500 on B
12  COMM T3
13    START T5
14    r T5 A =800
15  COMM T4      // joe commits
16    r T5 B =1400
17    r T5 C =1000

T1 commit
T2 commit
T3 commit
T4 commit
T5 active

101 A 800 (T1 commit)
102 B 900 (T2 commit)
103 A 300 (T4 commit) [-> 101]
104 C 1000 (T3 commit)
105 B 1400 (T4 commit) [-> 102]

```

and

- at 14, the banker reads the last committed value of "A" which is still 800 (T4 did not yet commit)
- at 16, the banker reads the last committed value of "B", which is now 1400, since T4 did commit
- he then reads C, 1000

so our banker now believes the total is 800+ 1400+ 1000 (no garbage collection in this example for clarity).

3.2 - Snapshot transaction

The solution is to freeze the situation when the banker starts his total. This is precisely what the snapshot isolation mode does: it saves the state of the other transactions when it starts. So no matter how many updates, deletes, commits or rollbacks are performed after the snapshot starts, a read in the context of the snapshot transaction will always return the value which was committed at the time the snapshot started.

Here is an example (script 26):

```
01 START T1
02 c T1 A 800
03 COMM T1
04  START T2
05  c T2 B 900
06  COMM T2
07    START T3
08    START T4
09    u T4 A 300
10    c T3 C 1000
11    u T4 B 1400
12    COMM T3
13      START T5 SNAP NO_W RW // o=4
14      r T5 A =800
15      COMM T4
16      r T5 B =900
17      r T5 C =1000

T1 commit
T2 commit
T3 commit
T4 commit
T5 snap  active

101 A  800 (T1  commit)
102 B  900 (T2  commit)
103 A  300 (T4  commit) [-> 101]
104 C 1000 (T3  commit)
105 B 1400 (T4  commit) [-> 102]
```

And

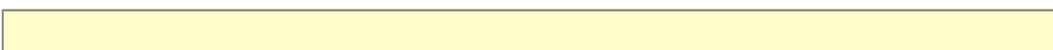
- when T5 reads the amount of "B", it reads the value this account had when T5 started. Since T4 had not committed, the previous committed value of "B" is 900, so the total is now correct.

3.3 - Snapshot implementation

Snapshot is implemented by attaching to the snapshot transaction a copy of the transaction list.

Let's assume two other customer, "D" and "E" had performed and completed their operations long before our banker's summing operation.

Here is the picture with the snapshot's copy of the transaction list when it started (script 27):



```

01 START T1 RC
02 c T1 D 330
03 COMM T1
04  START T2 RC
05  c T2 E 440
06  COMM T2
07  START T3 RC
08  c T3 A 800
09  COMM T3
10  START T4 RC
11  c T4 B 900
12  COMM T4
13  START T5 RC
14  START T6 RC // joe
15  u T6 A 300 // joe
16  c T5 C 1000
17  u T6 B 1400 // joe
18  COMM T5
19  START T7 SNAP % T1 rd_com commit
.      % T2 rd_com commit
.      % T3 rd_com commit
.      % T4 rd_com commit
.      % T5 rd_com commit
.      % T6 rd_com active
.      % T7 snap active
20  r T7 A =800
21  COMM T6 // joe
22  START T8 RC
23  d T8 B
24  COMM T8
25  r T7 B =900
26  r T7 C =1000

```

```

T1 rd_com commit
T2 rd_com commit
T3 rd_com commit
T4 rd_com commit
T5 rd_com commit
T6 rd_com commit
T7 snap active
T8 rd_com commit

```

```

101 D 330 (T1 rd_com commit)
102 E 440 (T2 rd_com commit)
103 A 800 (T3 rd_com commit)
104 B 900 (T4 rd_com commit)
105 A 300 (T6 rd_com commit) [-> 103]
106 C 1000 (T5 rd_com commit)
107 B 1400 (T6 rd_com commit) [-> 104]
108 B -del (T8 rd_com commit) [-> 107]

```

In this case

- at 20 the banker requests the value of "A". The last row version is "105 A 300 (T6)". In the snapshot, T6 is still active, so this is not the reading of a committed value. So the previous value is returned, "103 A 800 (T3)
- at 25, "B" is requested
 - the last row version is "108 deleted (T8)". Since T8 started after the T7 snapshot, we have to look for a previous row version
 - the previous "107 B 1400 (T6)" was modified by T6 before our snapshot started, and is committed at the time of the read. But the copy of the transactions when the snapshot started tells that T6 was not committed when the snapshot started ("% T6 active").
 - so finally the previous version "104 B 900 (T4)" is read

A couple of remarks

- our example shows that the snapshot will correctly read values from rows which are deleted and committed if this removal was performed by a transaction started after the snapshot start
- it is also obvious that the copy of all the transaction list when the snapshot starts is not required. Only the transactions which are still active matter: their modifications could be committed before the snapshot reads their values.
- at the same time, the snapshot modifies the previously outlined garbage collection rules:
 - when 25 reads "B", it should garbage collect the "B" key
 - the last committed version of "B" is "108 B delete (T8)"
 - but we cannot remove all previous versions of "B", since the snapshot has a committed version of "B" in its T4 transaction

So this brings us to the whole OIT OAT OST business.

3.4 - OAT, OAST, OATOOAST, Oh what else

To compute which transactions must be copied into a snapshot's private transaction list, it is easy to see that all row versions committed when a snapshot transaction starts can no longer be modified (and committed) after the snapshot start. The snapshot private transaction list is only there to allow the consistent read of row versions which might be modified and committed by a transaction active when the snapshot starts.

In the previous example, if the banker wants to read "D", using the global transaction list or the private transaction list will return the same values, "101 D 330 (T1)". This value has been committed before the snapshot started. This is the last committed value of "D", and this is the value that the snapshot should read, even if some active transaction at the time of the

snapshot start modifies or deletes it.

This is also true for the "102 E 440 (T2)", "103 A 800 (T3)", "104 B 900 (T4)" and "106 C 1000 (T5)" row versions.

Therefore, when a snapshot starts, we only must include transactions not active at this time. This transaction is called the OAT (**O**ldest **A**ctive **T**ransaction).

Here is the situation using the OAT (script 28):

```
01 START T1 RC
02 c T1 D 330
03 COMM T1
04  START T2 RC
05  c T2 E 440
06  COMM T2
07    START T3 RC
08    c T3 A 800
09    COMM T3
10    START T4 RC
11    c T4 B 900
12    COMM T4
13      START T5 RC
14      START T6 RC // joe
15      u T6 A 300 // joe
16      c T5 C 1000
17      u T6 B 1400 // joe
18      COMM T5
19        START T7 SNAP % T6 rd_com active
.          % T7 snap active
20        r T7 A =800
21        COMM T6 // joe
22          START T8 RC
23          d T8 B
24          COMM T8
25          r T7 B =900
26          r T7 C =1000

T1 rd_com commit
T2 rd_com commit
T3 rd_com commit
T4 rd_com commit
T5 rd_com commit
T6 rd_com commit
T7 snap active
T8 rd_com commit

101 D 330 (T1 rd_com commit)
102 E 440 (T2 rd_com commit)
```

```

103 A 800 (T3 rd_com commit)
104 B 900 (T4 rd_com commit)
105 A 300 (T6 rd_com commit) [-> 103]
106 C 1000 (T5 rd_com commit)
107 B 1400 (T6 rd_com commit) [-> 104]
108 B -del (T8 rd_com commit) [-> 107]

```

For garbage collection purposes, the threshold is the OATOOAST

- garbage is performed when a key is read.
- we cannot remove any row that some snapshot still requires for performing a consistent read.

The threshold is the "oldest active transaction of the oldest active snapshot transaction".

Lets take the following situation (script 29):

```

01 START T1 RC
02 c T1 A 800
03 COMM T1
04  START T2 RC
05  u T2 A 811
06  COMM T2
07  START T3 RC
08  c T3 B 950
09  COMM T3
10  START T4 RC
11  u T4 A 822
12  START T5 SNAP  % T4 rd_com active
.      % T5 snap active
13  START T6 RC
14  u T6 B 955
15  COMM T4
16  START T7 SNAP % T5 snap active
.      % T6 rd_com active
.      % T7 snap active
17  -garb T1 A 101
18  r T5 A =811

T1 rd_com commit
T2 rd_com commit
T3 rd_com commit
T4 rd_com commit OATOOAST
T5 snap active OAT=T4
T6 rd_com active
T7 snap active OAT=T5
---OAST=T5, OAT=T5, OATOOAST= T4

101 A 800 (T1 rd_com commit) G

```

```
102 A 811 (T2 rd_com commit)
103 B 950 (T3 rd_com commit)
104 A 822 (T4 rd_com commit) [-> 102]
105 B 955 (T6 rd_com active) x [-> 103]
```

and

- when the snapshot T5 starts, it builds the list of active transactions at that time, T4 and T5
- when the snapshot T7 starts, it builds the private list T5, T6 and T7
- now action 18 performs a read. It tries to garbage the stale row versions. Even if T4 has committed at that time, it cannot remove previous versions of "A", since the snapshot T5 is still active, and when T5 started, T4 was active and the snapshot cannot read values of an transaction which was active when it started.

So we can only garbage collect rows whose transaction are older than the "oldest active snapshot's (T5) oldest active transaction (T4)", which could be called something like OATOOAST "oldest active transaction of the oldest snapshot transaction"

When the garbage collector starts, all linked row versions of key "A" are (display 30):

```
101 A 800 (T1 rd_com commit)
102 A 811 (T2 rd_com commit) [-> 101])
104 A 822 (T4 rd_com commit) [-> 102]
```

and

- "104 A 822 (T4)" cannot be removed because it's transaction T4 is the OATOOAST
- "102 A 811 (T2)" is committed and must be kept around as the last committed value later than the OAOA. This is the value that can be read by snapshot T5
- "101 A 800 (T1)" will never be read by anybody and can safely be removed (as our figure demonstrates)

For modifications performed by the snapshot, they behave like the read-committed modifications if the other connection did not commit : if a lock was placed on the key by another key, the modification triggers an exception, like this (script 31):

```
01 START T1 RC
02 c T1 A 800
03 COMM T1
04 START T2 SNAP // o=2
05 START T3 RC
06 u T3 A 900
07 u T2 A 1000 *** lock_ver 102
```

```

08  r T2 A =800

T1 rd_com  commit
T2 snap   active OAT T2
T3 rd_com  active

101 A 800 (T1 rd_com  commit)
102 A 900 (T3 rd_com  active) x [-> 101]

```

is refused (Delphi exception)

If another transaction modifies a row after the snapshot started, any modification of this row by the snapshot also triggers an exception (script 32):

```

01 START T1 RC
02 c T1 A 800
03 COMM T1
04  START T2 SNAP // o=2
05  START T3 RC
06  u T3 A 1000
07  COMM T3
08  r T2 A =800
09  u T2 A 1100 * prev_commit_modif 102
10  r T2 A =800

T1 rd_com  commit
T2 snap   active OAT T2
T3 rd_com  commit

101 A 800 (T1 rd_com  commit)
102 A 1000 (T3 rd_com  commit) [-> 101]

```

If the snapshot updates a key, it puts a lock on it, and if another transaction tries to modify this key we get the usual modification of a locked key exception (script 33):

```

01 START T1 RC
02 c T1 A 800
03 COMM T1
04  START T2 SNAP // o=2
05  u T2 A 1100
06  r T2 A =1100
07  START T3 RC
08  u T3 A 1000 *** lock_ver 102
09  r T3 A =800
10  COMM T3

```

```

11  r T2 A =1100

T1 rd_com  commit
T2 snap   active OAT T2
T3 rd_com  commit

101 A 800 (T1 rd_com  commit)
102 A 1100 (T2 snap   active OAT T2) x [-> 101]

```

The snapshot is not allowed to update a key updated by a transaction started before it started, even if this previous update is committed (script 34):

```

01 START T1 RC
02 c T1 A 802
03 COMM T1
04  START T2 SNAP (o=2)
05  r T2 A =802
06  u T2 A 804
07  START T3 SNAP (o=2)
08  u T3 A 810 *** lock_ver 102
09  u T2 A 813
10  r T2 A =813
11  COMM T2
12  r T3 A =802
13  u T3 A 814 *** snap_prev_upd 103
14  r T3 A =802

T1 rd_com  commit
T2 snap   commit OAT T2
T3 snap   active OAT T2
-- oat=T3 oast=T3 oatooast=T2

101 A 802 (T1 rd_com  commit)
102 A 804 (T2 snap   commit OAT T2) [-> 101]
103 A 813 (T2 snap   commit OAT T2) [-> 102]

```

where

- action 13 triggers an exception (even if T2 is committed)

The behaviour of delete is similar to the update (script 35):

```

01 START T1 RC
02 c T1 A 800
03 COMM T1

```

```

04 START T2 SNAP // o=2
05  START T3 RC
06  d T3 A
07  COMM T3
08  r T2 A =800
09  u T2 A 1100 * prev_commit_modif 102
10  r T2 A =800

T1 rd_com  commit
T2 snap   active OAT T2
T3 rd_com  commit

101 A 800 (T1 rd_com  commit)
102 A -del (T3 rd_com  commit) [-> 101]

```

3.5 - How do the OAT and OATOOAST move ?

So basically, we must keep track of (or compute on the fly) the following values:

- the next transaction number
- the OAT which is used to build a snapshot's private transaction list
- each snapshot remembers the value of the OAT (oldest active) when it starts
- the system maintains the OAST (oldest active snapshot)
- together those produce the OATOOAST (the OAT of the OAST), and this is used by the garbage collector to keep the potentially useful row versions around

The OAT moves forward (script 36)

- when no transaction was active and a new transaction starts
- when the transaction which was the OAT commits or is rolled back

```

01 START T1 RC           // o 1  <= new active
02 c T1 A 800
03 COMM T1              // o -
04  START T2 RC         // o 2  <= new active
05  c T2 B 900
06  COMM T2            // o -
07    START T3 RC       // o 3  <= new active
08    START T4 RC       // o 3
09    u T4 A 300
10    c T3 C 1000
11    u T4 B 1400
12    COMM T3           // o 4    <= OAT commits
13      START T5 SNAP (o=4) // o 4
14      r T5 A =800

```

```

15  ROLL T4          // o 5    <= OAT rolls back
16      START T6 SNAP (o=5) // o 5
17  r T5 B =900
18      COMM T6      // o 5    <= OAT commits
18      START T7 SNAP (o=5) // o 5
19      COMM T6      // o 5
20  r T5 C =1000
21      COMM T5      // o 7    <= OAT commits

```

The OAST moves forward (script 37)

- when no snapshot transaction was active and a new snapshot transaction starts
- when the transaction which was the OAT commits or is rolled back

```

01 START T1 RC          // o 1 -
02 c T1 A 800
03 COMM T1              // o - -
04  START T2 RC        // o 2 -
05  c T2 B 900
06  COMM T2            // o - -
07  START T3 RC        // o 3 -
08  START T4 RC        // o 3 -
09  u T4 A 300
10  c T3 C 1000
11  u T4 B 1400
12  COMM T3            // o 4 -
13  START T5 SNAP (o=4) // o 4 5    <= new oldest snap
14  r T5 A =800
15  ROLL T4            // o 5 5
16  START T6 SNAP (o=5) // o 5 5
17  r T5 B =900
18  START T7 SNAP (o=5) // o 5 5
19  COMM T6            // o 5 5
20  r T5 C =1000
21  COMM T5            // o 7 7    <= OAST commits

```

The OATOAST moves at the same time as the OAST (since it is the property of the OAST)
(script 38)

```

01 START T1 RC          // o 1 - -
02 c T1 A 800
03 COMM T1              // o - - -

```

```

04 START T2 RC           // o 2 - -
05 c T2 B 900
06 COMM T2              // o - - -
07  START T3 RC         // o 3 - -
08    START T4 RC       // o 3 - -
09      u T4 A 300
10      c T3 C 1000
11      u T4 B 1400
12 COMM T3              // o 4 - -
13  START T5 SNAP (o=4) // o 4 5 4
14    r T5 A =800
15  ROLL T4             // o 5 5 4
16    START T6 SNAP (o=5) // o 5 5 4
17      r T5 B =900
18        START T7 SNAP (o=5) // o 5 5 4
19          COMM T6      // o 5 5 4
20            r T5 C =1000
21          COMM T5      // o 7 7 5

```

As a final touch, we can also remove from a snapshot's private transaction list the snapshot itself, since each transaction is always allowed to read its own modifications. Keeping "% T5 snap active" in T5's private list or "% T6 snap active" etc are not necessary.

3.6 - Why is OAT important

OAT is used to compute the size of each snapshot's private transaction list, which is evaluated when a snapshot starts

current- 1- OAT

Obviously the less time we keep a transaction active, the shorter all the private transaction lists will be, if we use snapshot transactions. Keeping long private transaction lists occupies memory and take time to analyze.

3.7 - Why is OATOOAST important

The OATOOAST defines the limit after which no garbage collection can be done.

Garbage collection is performed for each key separately, by traveling the linked lists of previous row versions. For some list, we can only start removing versions if the row version is lower than the OATOOAST. Starting from this row version

- we can remove all rollbacks
- if a delete is found we can remove everything earlier

- else we keep the first committed, and remove everything earlier

Keeping the OATOOAST as close to the most recent active transaction as possible will allow to remove the unused record versions. This will reduce disk space, improve grouping of record versions in disc page, reduce disc reads.

4 - The Delphi Transaction simulator

4.1 - Overview of the project

Basically we defined the actions, which were used by the presentation above.

Those actions are

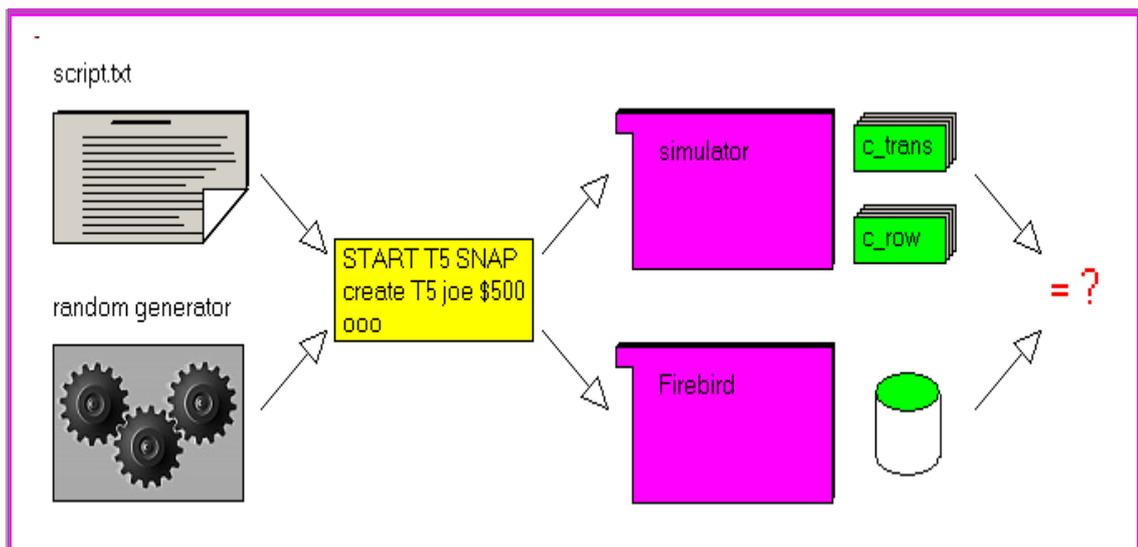
- either read from a .TXT script
- or are generated randomly by a generator

The script engine or the generator process those actions using two channels

- the Delphi code simulating Firebird's behaviour
- the Firebird Sql engine directly by using lbx data access components

The result of our simulator can be compared to the Firebird results.

Here is an overview of the process:



4.2 - The Delphi Classes

We first started by writing the basic simulator, using manually written scripts and visually checking the result (is the value read correct, should we be able to update this row etc).

To make sure that bug correction would not invalidate previous scripts, we added a comparison of a script with a manually checked version of this script. We can run all scripts automatically, thereby performing some kind of regression tests.

Then we created the random generator to be sure we covered as many cases as possible.

Since some corner cases still were unclear, we then added the Firebird part. Checking the Firebird output with our simulator output allowed to fix some bugs. For scripts, this is of course better than our manually checked scripts.

We then added some utilities, like pretty printing the output or renumbering the transaction to be able to turn random scripts into .txt scripts for detailed manual analysis.

The simulator is based on two **Classes**, the transaction list and the row version lists.

Here is the transaction definition:

```
t_isolation_type= (e_unknown_isolation,
    e_snapshot_isolation, e_read_committed_isolation);
t_transaction_state_type= (e_unknown_state,
    e_active_state,
    e_committed_state, e_rolled_back_state,
    e_limbo_state); // failure during multi database 2 phase commit
c_transaction=
    Class(c_basic_object)
        m_transaction_id: Integer;

        m_transaction_isolation: t_isolation_type;
        m_transaction_state: t_transaction_state_type;

        m_read_only: Boolean;
        m_wait: Boolean;

        m_modification_count: Integer;
        // -- for comping the OST
        m_transaction_snapshot_number: Integer;
```

```

m_c_snapshot_transaction_list_ref: c_transaction_list;

m_c_oldest_active_when_started: c_transaction;

// -- still has a lock on itself. Not implemented
m_alive: Boolean;

// -- since "rolled back" are converted into "committee", remember
this
m_mark_for_rollback_to_commit_optimisation: Boolean;

// -- for simulation, keep this one alive at least this long
m_long_transaction_life_min, m_transaction_iteration_start:

Constructor create_transaction(p_name: String;
    p_transaction_id: Integer; p_isolation_type: t_isolation_type;
    p_wait: Boolean; p_read_only: Boolean);
Function f_c_self: c_transaction;
Function f_display_transaction: String;

Procedure commit;
Procedure commit_retaining;
Procedure roll_back;

Function f_is_alive: Boolean;

Function f_c_clone: c_transaction;

Destructor Destroy; Override;
End; // c_transaction

```

and the transaction list, with the next transaction id, the creation of a transaction and several helpers to find the OAT etc

```

c_transaction_list=
// -- all transactions since created or last backup/restore
Class(c_basic_object)
    m_c_transaction_list: tList;

    m_next_transaction_id: Integer;

Constructor create_transaction_list(p_name: String);

Function f_c_self: c_transaction_list;

```

```

Function f_transaction_count: Integer;
Function f_c_transaction(p_transaction_index: Integer):
c_transaction;

Function f_c_snapshot_transaction(p_transaction_id: Integer):
c_transaction;

Function f_c_start_transaction(p_isolation_type: t_isolation_type;
p_wait: Boolean; p_read_only: Boolean): c_transaction;

Function f_c_oldest_not_committed_nor_rolledback_transaction:
c_transaction;

Function f_c_oldest_active_transaction: c_transaction;
Function f_c_oldest_active_snapshot_transaction: c_transaction;
Function
f_c_oldest_active_snapshot_oldest_active_when_started_transactio
n: c_transaction;

Function f_can_collect_garbage_before_id: Integer;

Function f_is_transaction_in_list(p_transaction_id: Integer):
Boolean;

Procedure toggle_rolled_back_to_committed;

Destructor Destroy; Override;
End; // c_transaction_list

```

The version rows are defined by:

```

c_row=
Class(c_basic_object)
m_row_id: Integer;
m_account: String;
m_amount: Integer;

m_c_transaction_ref: c_transaction;

m_is_locked: Boolean;
m_is_deleted: Boolean;

```

```

m_is_garbage: Boolean;

// -- chains the successive row versions
m_c_previous_row_version: c_row;
// -- used only to shortcut a row during garbage collection
m_c_next_row_version: c_row;

Constructor create_row(p_name: String;
    p_row_id: Integer;
    p_account: String; p_amount: Integer);
Function f_c_self: c_row;
End; // c_row

c_table=
Class(c_basic_object)
    m_c_row_list: tList;
    m_next_row_id: Integer;

    m_remove_key_garbage_versions: Boolean;

Constructor create_table(p_name: String);

Function f_row_count: Integer;
Function f_c_row(p_row_index: Integer): c_row;

Function f_c_last_non_garbage_row_by_account(p_account:
String): c_row;
Function
f_c_last_non_garbage_non_deleted_row_by_account(p_account:
String): c_row;

Function f_c_locked_row(p_account: String): c_row;

Procedure mark_row_as_garbage(p_triggering_id: Integer;
p_c_to_garbage_row: c_row);
Function f_remove_key_garbage_versions(p_triggering_id:
Integer;
    p_c_row: c_row;
    p_garbage_start_id: Integer;
p_toggle_rollback_transaction_to_committed: Boolean;
    Var pv_garbage_error: t_table_error_type): Boolean;

Procedure unlock_rows(p_c_transaction: c_transaction);

```

```

Function f_c_append_row(p_c_transaction_ref: c_transaction;
p_account: String;
    p_amount: Integer): c_row;

Function
f_is_deleted_and_committed(p_c_last_non_garbage_row_by_account: c_row): Boolean;

Function
f_c_committed_modification_after_snapshot(p_c_last_row: c_row;
    p_c_transaction: c_transaction): c_row;

Function f_test_available_row(p_c_transaction_ref: c_transaction;
p_account: String): Boolean;

// -- CRUD
Function f_c_create_row(p_c_transaction_ref: c_transaction;
p_account: String;
    p_amount: Integer, Var pv_table_error_type: t_table_error_type):
c_row;
Function f_c_read_row(p_c_transaction_ref: c_transaction;
p_account: String;
    Var pv_amount, pv_garbage_start_id: Integer,
    Var pv_table_error_type: t_table_error_type): c_row;
Function f_c_update_row(p_c_transaction_ref: c_transaction;
p_account: String;
    p_amount: Integer, Var pv_table_error_type: t_table_error_type):
c_row;
Function f_c_delete_row(p_c_transaction_ref: c_transaction;
p_account: String;
    Var pv_table_error_type: t_table_error_type): c_row;

Destructor Destroy; Override;
End; // c_table

```

A *c_database* container **Class** is also defined:

```

c_database=
Class(c_basic_object)
    m_c_transaction_list: c_transaction_list;
    m_c_account: c_table;

```

```

Constructor create_database(p_name: String);

Procedure sweep_database(p_transaction_id_max: Integer;
    Var pv_sweep_error: t_table_error_type);

Destructor Destroy; Override;
End; // c_database

```

The *c_action_memory_manager* receives the actions from the script reader or the simulator and manages the *c_transaction* and *c_row*:

```

c_action_memory_manager=
Class(c_basic_object)
    m_c_database_ref: c_database;

Constructor create_action_memory_manager(p_name: String);

Function f_action_start_transaction(p_transaction_isolation:
t_isolation_type;
    p_wait, p_read_only: Boolean): Integer; Virtual;
Procedure action_commit(p_transaction_id: Integer); Virtual;
Procedure action_rollback(p_transaction_id: Integer); Virtual;

Procedure action_create_row(p_transaction_id: Integer; p_key:
String;
    p_amount: Integer); Virtual;
Procedure action_read_row(p_transaction_id: Integer; p_key:
String;
    Var pv_amount: Integer); Virtual;
Procedure action_update_row(p_transaction_id: Integer; p_key:
String;
    p_amount: Integer); Virtual;
Procedure action_delete_row(p_transaction_id: Integer; p_key:
String); Virtual;

Procedure action_sweep;

Destructor Destroy; Override;
End; // c_action_memory_manager

```

And the script interpreter analyzes the .TXT scripts and calls the actions

```
c_script_interpreter=  
Class(c_basic_object)  
  m_c_action_memory_manager_ref: c_action_memory_manager;  
  
  m_c_script_stringlist: c_stringlist;  
  
  m_script_step_by_step: Boolean;  
  m_step_by_step_next_index: Integer;  
  
Constructor create_script_interpreter(p_name: String);  
Procedure load_and_interpret(p_path, p_file_name: String);  
Destructor Destroy; Override;  
End; // c_script_interpreter
```

For random simulation we use this **Class**:

```
c_random_action_interpreter=  
Class(c_basic_object)  
  m_transaction_weight: Double;  
  m_transaction_create_weight: Double;  
  m_transaction_snapshot_weight: Double;  
  m_transaction_wait_weight: Double;  
  m_transaction_read_write_weight: Double;  
  m_long_transaction_weight: Double;  
  m_long_transaction_size: Integer;  
  m_transaction_commit_weight: Double;  
  m_transaction_rollback_weight: Double;  
  m_row_weight: Double;  
  m_row_create_weight: Double;  
  m_row_read_weight: Double;  
  m_row_update_weight: Double;  
  m_row_delete_weight: Double;  
  m_crash_weight: Double;  
  
  m_transaction_count_max: Integer;  
  
  // -- operation  
  m_iteration: Integer;
```

```

m_c_row_account_list: tStringList;
m_simulation_amount: Integer;

m_c_action_memory_manager_ref: c_action_memory_manager;

m_step_by_step: Boolean;

// -- stats
m_transaction_total: Double;
m_row_total: Double;

m_transaction_create_count_ok, m_transaction_create_count_pb:
Integer;
    m_transaction_snapshot_count_ok,
m_transaction_snapshot_count_pb: Integer;
    m_transaction_wait_count_ok, m_transaction_wait_count_pb:
Integer;
    m_transaction_read_write_count_ok,
m_transaction_read_write_count_pb: Integer;
    m_transaction_commit_count_ok,
m_transaction_commit_count_pb: Integer;
    m_transaction_rollback_count_ok,
m_transaction_rollback_count_pb: Integer;

m_row_create_count_ok, m_row_create_count_pb: Integer;
m_row_read_count_ok, m_row_read_count_pb: Integer;
m_row_update_count_ok, m_row_update_count_pb: Integer;
m_row_delete_count_ok, m_row_delete_count_pb: Integer;

m_active_transaction_count, m_active_transaction_trial_count:
Integer;

Constructor create_random_action_interpreter(p_name: String);

Procedure start_simulation;
Procedure stop_simulation;

Destructor Destroy; Override;
End; // c_random_action_interpreter

```

On the Firebird side, we have a `c_action_firebird_manager` with the same methods as the `c_action_memory_manager`

```

c_action_firebird_manager=
  Class(c_action_memory_manager)
    m_c_firebird_database_2_ref: c_firebird_database_2;

  Constructor create_action_firebird_manager(p_name: String);

  Function f_action_start_transaction(p_transaction_isolation:
t_isolation_type;
    p_wait, p_read_only: Boolean): Integer; Override;
  Procedure action_commit(p_transaction_id: Integer); Override;
  Procedure action_rollback(p_transaction_id: Integer); Override;

  Procedure action_create_row(p_transaction_id: Integer, p_key:
String;
    p_amount: Integer); Override;
  Procedure action_read_row(p_transaction_id: Integer, p_key:
String;
    Var pv_amount: Integer); Override;
  Procedure action_update_row(p_transaction_id: Integer, p_key:
String;
    p_amount: Integer); Override;
  Procedure action_delete_row(p_transaction_id: Integer, p_key:
String); Override;

  Destructor Destroy; Override;
End; // c_action_firebird_manager

```

but instead of modifying the *c_transaction* or *c_row*, this **Class** directly modified a Firebird database:

```

c_firebird_database_2=
  Class(c_basic_object)
    m_ip, m_path, m_file_name: String;

    m_c_ib_database: tlbDatabase;
    m_c_ib_sql: tlbSql;
    m_c_ib_dataset: tlbDataSet;

    m_c_ib_transaction_list: tList;

    // -- simulate a version number
    m_fake_version_id: Integer;

```

m_rows_affected: Integer;

Constructor *create_firebird_database_2(p_name, p_ip, p_path, p_file_name: String);*

Function *f_c_add_ib_transaction(p_snapshot: Boolean): tlbTransaction;*

Function *f_c_ib_transaction(p_transaction_index: integer): tlbTransaction;*

Function *f_execute_ibsql(p_c_ib_transaction: tlbTransaction; p_request: String): Boolean;*

Function *f_open_ibdataset(p_c_ib_transaction: tlbTransaction; p_request: String): Boolean;*

Function *f_next_fake_version_id: Integer;*

Procedure *commit_all_transactions;*

Function *f_ib_transactions_state: String;*

Function *f_start_transaction(p_snapshot: Boolean): Boolean;*

Function *f_commit_transaction(p_transaction_id: Integer): Boolean;*

Function *f_rollback_transaction(p_transaction_id: Integer):*

Function *f_insert_account(p_transaction_id: Integer; p_name: String; p_amount: Integer): Boolean;*

Function *f_read_account(p_transaction_id: Integer; p_name: String): Integer;*

Function *f_update_account(p_transaction_id: Integer; p_name: String;*

p_amount: Integer): Boolean;

Function *f_delete_account(p_transaction_id: Integer; p_name: String): Boolean;*

Destructor *Destroy; Override;*

End; *// c_firebird_database_2*

So we have 4 available pathes :

script.txt -> c_script_interpreter -> c_action_memory_manager ->

```
c_transaction / c_row  
  script.txt -> c_script_interpreter -> c_action_firebird_manager ->  
c_firebird_database_2  
  c_random_action_generator -> c_action_memory_manager ->  
c_transaction / c_row  
  c_random_action_generator -> c_action_firebird_manager ->  
c_firebird_database_2
```

4.3 - The methods

Beyond the structures, it would be interesting to present here the method code.

Much of the code like reading a .TXT to extract the action parameters is quite boring.

Implementing the CRUD actions is more interesting, but this code is quite intricate to cope with all the tests and checks, plus many sanity checks we performed. And it is possible that some of our tests are redundant.

For instance, for updating a row:

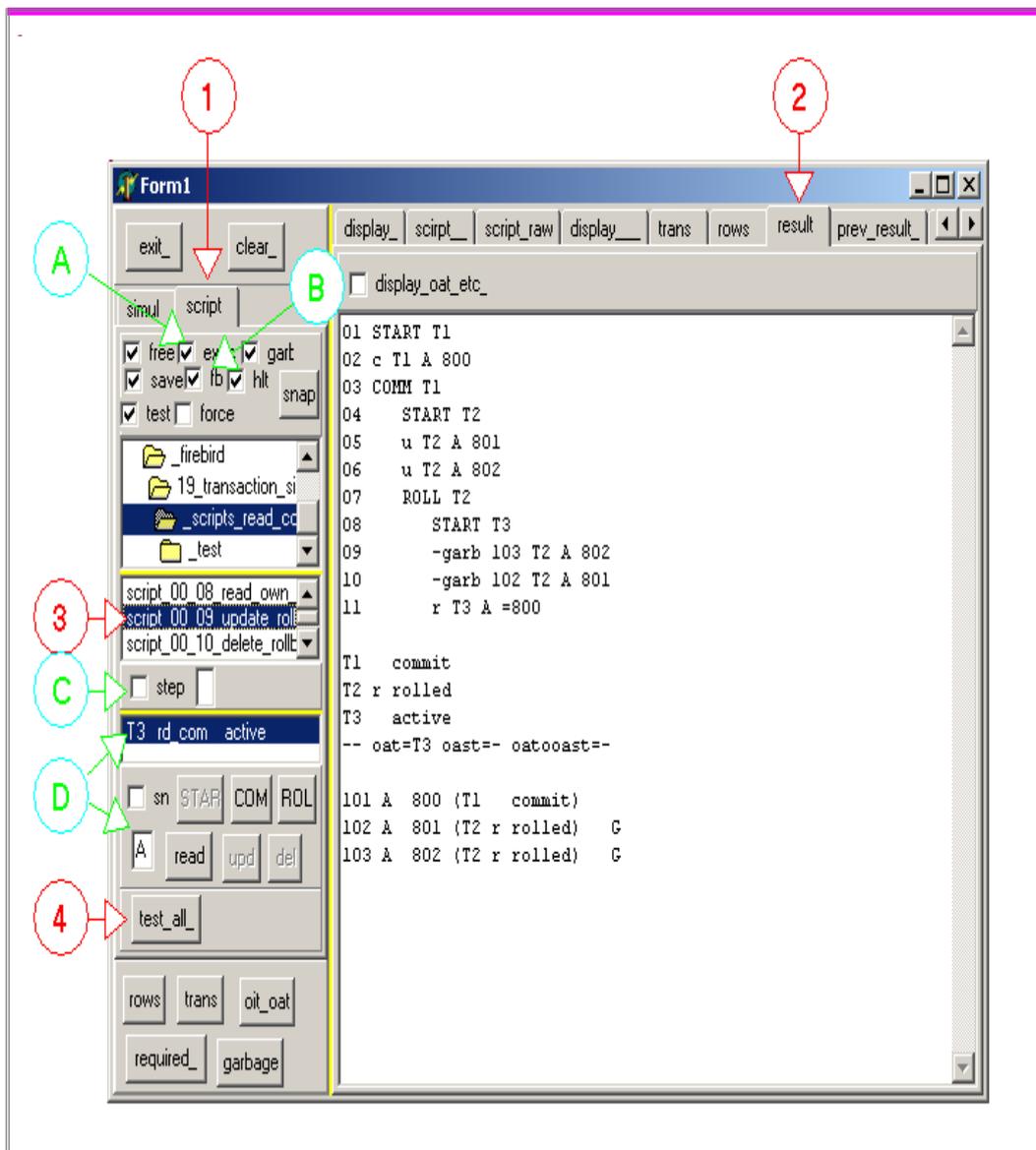
```
// -- check that the transaction is not read-only  
// -- check that this transaction can "read" a previous version  
// -- check if some other transaction has placed a lock  
// -- if in snapshot, we cannot update a previously committed  
transaction if  
  // -- that transaction started before the shapshot  
  // -- check that this transaction did not delete the row in a previous  
action  
  
// -- now create the new row version
```

Anyway, the .ZIP source code contains the details

4.4 - Mini HowTo

We will just present the script and the random mode.

Here is an image of the program after a script has been interpreted:



To execute a script

- ✦ select the tabsheets 1 and 2
- ✦ select a script in the FileListBox 2. Clicking will execute the script, using the memory simulator and the Firebird database

To execute all scripts (better from the Windows explorer, or without the exceptions)

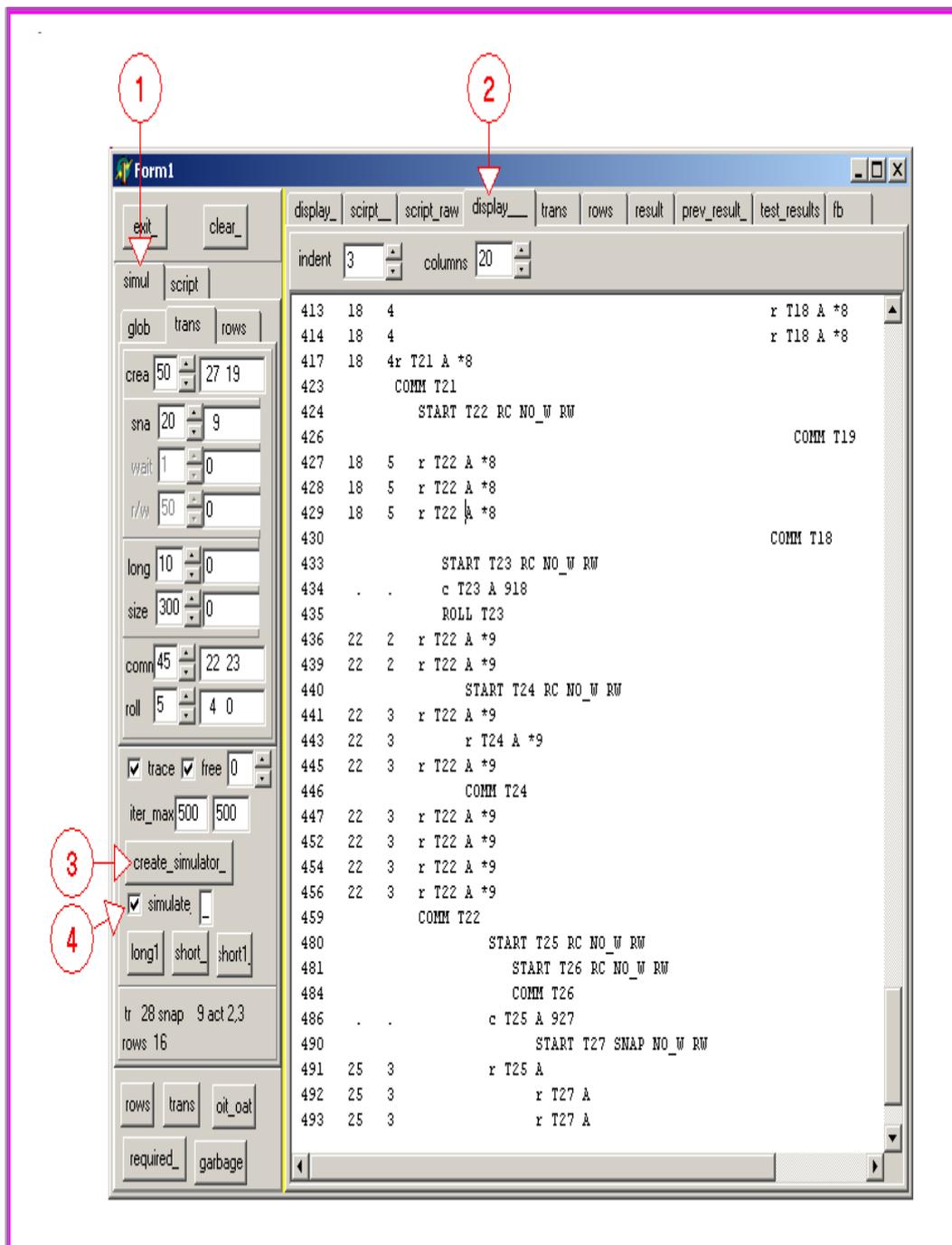
- ✦ click "test_all" 4

Some options

- to only display a script
 - ✦ uncheck "exec_" A,

-  select a script in the FileListBox 2
- to only use the memory simulation (no Firebird database)
 -  uncheck "fb_" B
- to execute step by step
 -  check "step" C
 -  select a script in the FileListBox 2
 -  ← the first action is executed and the focus is on the Edit at the right of "step"
 -  select the Edit at the right of "step" and type <space>, for on step, <return> to interpret without stepping
- to execute some more instructions after the step
 -  select an active transaction, if any, in the active transaction listbox D
 -  click "read", "COM", "ROLL"

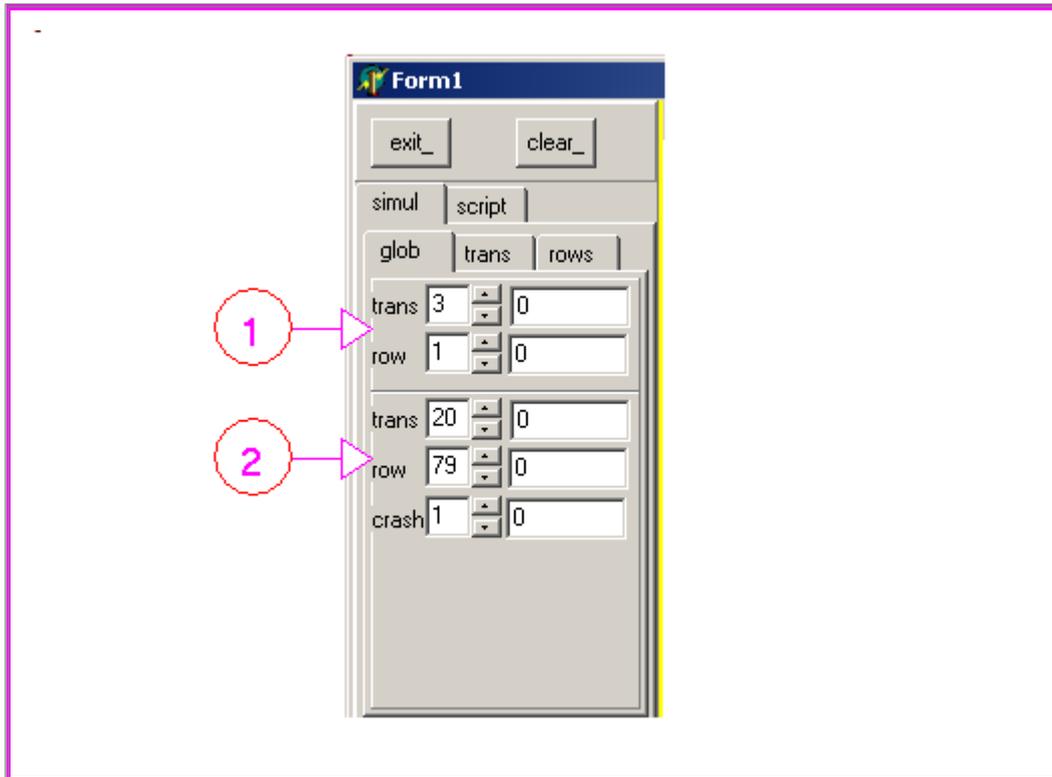
Here is the random action simulator:



To run a simulation

- 🔪 select tabsheets 1 and 2
- 🔪 create a new simulator 3
- 🔪 check "simulate_" 3
- ⬅ the focus is set to the Edit next to the "simulate_" checkbox
- 🔪 select this edit and type <space>, for on step, <return> to interpret without stepping

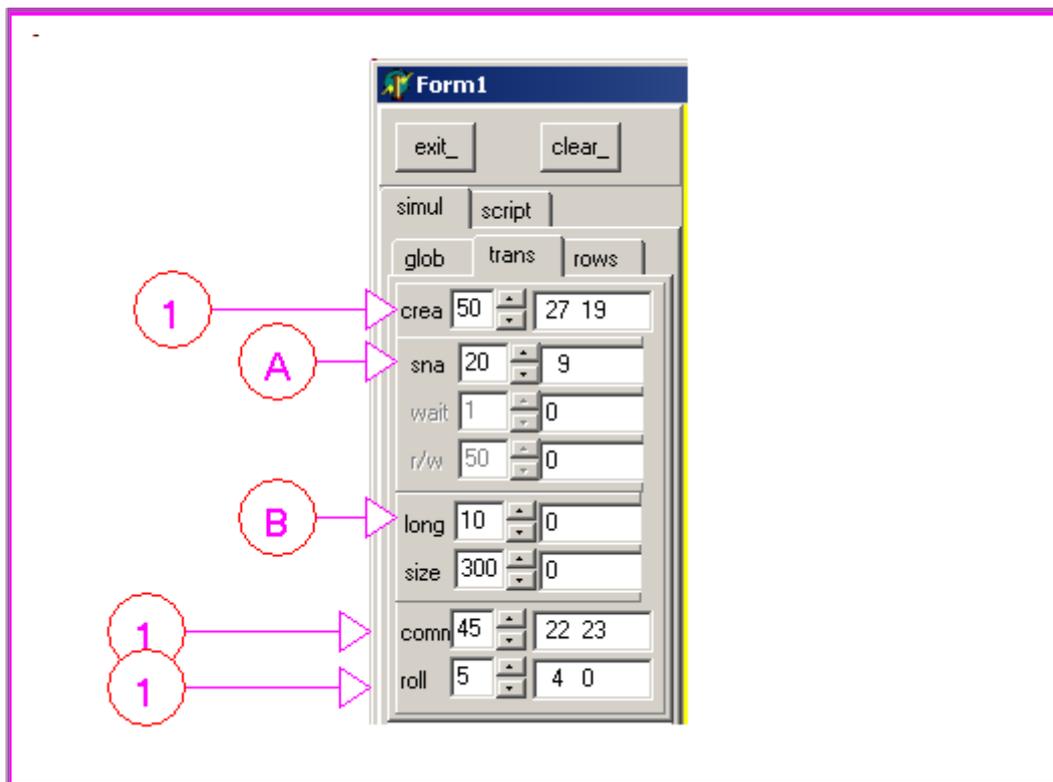
The random weights and some limits can be set by the controls above



and

- 1 sets the transaction count max and the account count max (here 3 transactions, and a single account "A")
- 2 sets the percentage of the random actions. In our case
 - 20 % transaction trials (start / commit / rollback)
 - 79 % row action trials (create, read, update, delete)
 - 1 % failure (dead transaction). This is currently not implemented

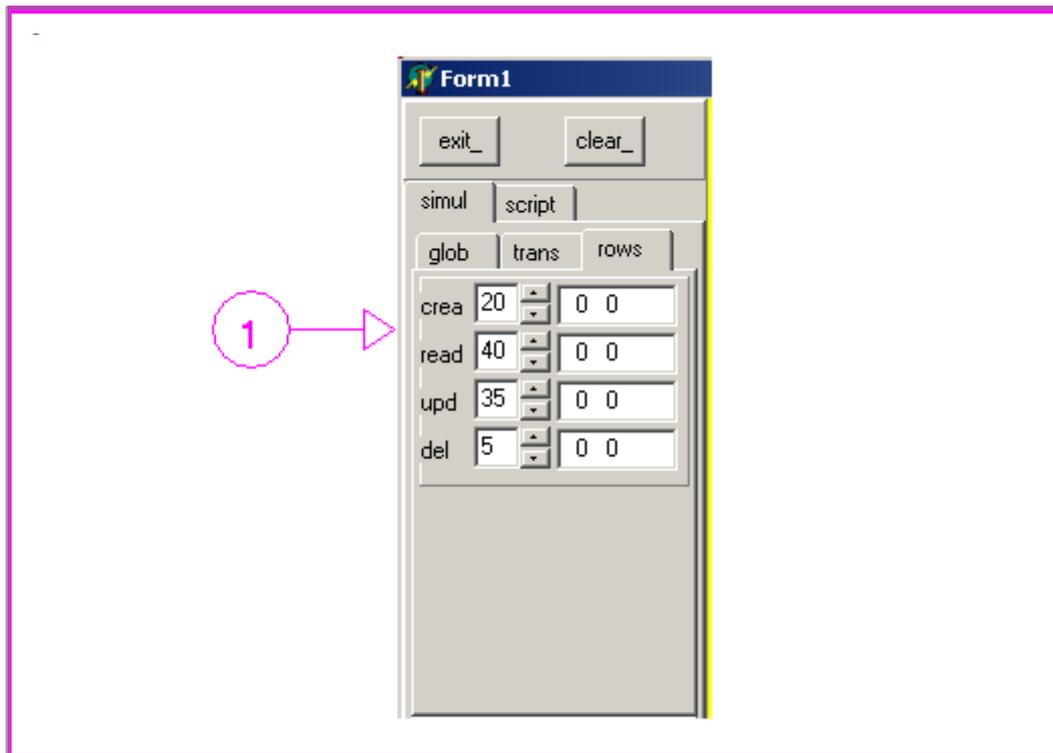
For the transactions:



and

- 1 sets the percentage of start / commit / rollback. Here
 - 50 % of creation
 - 45 % of commits
 - 5 % of rollbacks
- A sets the percentage of snapshot / read committed. In our case
 - 20 % of snapshots (therefore 80 % of read committed)
 wait vs no_wait and read vs read / write have not been implemented
- B sets the parameters of long transaction. In our case
 - 10 % of all transactions are "long transaction"
 - they should not be closed before 300 iterations elapsed

Finally for the rows:



where

- 1 sets the percentage among create / read / update / delete. In this picture:
 - 20 % create
 - 40 % read
 - 35 % update
 - 5 % delete

5 - Firebird Transactions

5.1 - Transaction isolation

Here are a couple of points about Interbase / Firebird

- we only implemented snapshot and read committed transactions.

In fact, there are 4 main possible modes

- snapshot, no wait
 - has the repeatable read property
 - if another transaction modified the row, an exception will be raised (we will not wait)
- snapshot, wait
 - has the repeatable read property
 - if another transaction modified the row, an exception will be raised, the

server will block our program (wait).

If and when the other party stops the transaction

- if he committed, we will receive an exception
- if he rolled back, our action will proceed
- read committed, no wait, record version
 - our transaction can read committed record versions, even if the other transaction committed after our transaction started
 - if another transaction modified the row,
 - if no wait, an exception will be raised (we will not wait)
- read committed, no record version
 - our transaction can read committed record versions, even if the other transaction committed after our transaction started
 - if another transaction modified the row,
 - if no wait, an exception will be raised (we will not wait)
 - if wait, our application is blocked until the other commits

There are additional modes, using table locks, mainly for maintenance purposes, which we will not present

5.2 - Read or Read / Write transactions

For the transactions, we can specify a read only or read / write mode.

The read only mode allows some optimisation : those transactions do not generate any record version. So, if the transaction rolls back, there is nothing to garbage collect from non existent record versions, and the transaction can directly considered as committed.

We did not implement this read-only option in our simulator

Each transaction also maintains an "update" flag. So even a read / write transaction for which no modification happened, could benefit from the direct switch from active to committed when rolled back.

We started to implement an *m_modification_count* in our simulator, but it is not used.

Also a read only transaction should not be taken into account when computing the "oldest active when the snapshot started". The read-only transaction does not modify (update or delete) any record value, so whether there are any read only transactions around or not should not influence the size of a snapshot's private transaction list.

OATOOAST should therefore become "oldest active read write transaction when the oldest active snapshot transaction started". OARWTWOASTS anybody ?

5.3 - Record Versions

The last record version of each key is complete, the next one are delta versions (depending on some size / optimization criterion)

This should be no concern for us

5.4 - OIT - Oldest Interesting Transaction

- each transaction can have 4 states, which means 2 bits.
- The state of all transactions is saved in the "transaction inventory page" (TIP) which, after some bookkeeping header contains 2 bits for each transaction.

This is cautiously saved to disc, so that if a crash occurs, the state of the transactions can be known.

Reading all those states is not necessary though. If we know that before some transaction id threshold they are all committed, it is not necessary to load the state of those transactions in memory.

- This magic level is called the "Oldest Interesting Transaction". The OIT is the oldest transaction whose state is other than committed.

So to know the state of some transactions, if the transaction id is

- lower or equal to the OIT we know they are committed
- greater than the OIT we must read those 2 bits

Maintaining the OIT value is simply a speed optimisation.

- when we rollback a transaction, if it did modify some rows (= if it generated some record versions), those record versions should be eliminated from the disc.

This is also an optimisation: the Server would work correctly, although more slowly, since it would have to perform more disc reads to find the relevant rows among the no longer used values.

- When is garbage collection performed ?

Garbage collection is performed :

- during a sweep
- when the server is started after a crash, to cleanup transaction which were active when the crash occurred
- otherwise, it depends on the Interbase / Firebird version:
 - for Interbase before version 7.1 SP1, the behaviour is the one presented here (before reading a row)
 - for Interbase 7.1 SP1 and later, when *rollback* is called, if the modified row count is less than 100.000, the modified rows are removed and the

transaction state directly set to committed. For more modifications than 100.000, the transaction is set to rolled back as before Ib 7.1.

- As explained above, since it often happens that many rows are not read regularly, their old record versions will not be removed before those reads.

Therefore Interbase / Firebird offer the "sweep" operation, which removes all garbage.

- sweep is
 - either launched manually by the DBA.
 - or triggered when a some garbage threshold is reached

To compute this threshold, we know that:

- all transaction older than the OIT are committed, by definition of the OIT
- record versions with transaction ids greater than the OATOOAST can still be used

Rows between OIT and $\text{Min}(\text{OATOOAST}, \text{OAT})$ can be garbage collected

And the default threshold for automatic garbage collection is 20.000. Sweep is triggered when

$$\text{Min}(\text{OATOOAST}, \text{OAT}) - \text{OIT} > 20.000$$

Let's also mention that the official name of our OATOOAST is "OST", and the documents quickly explain that for snapshots, "OST" is not, as one would believe "oldest snapshot transaction" but the "OAT when the Oldest Snapshot started".

- the automatic sweep is handled in a separate thread
- the sweep "rollback to committed optimization" : when a sweep is performed, all record versions which were modified by a rolled back transaction are removed. There is no longer any risk to find a record version referencing this transaction.

Interbase / Firebird then switches this transactions "rolled back" state to "committed".

The benefit of this is that the OIT might be increased (reducing the reading of the TIP pages etc).

We somehow fail why this toggling is useful. Why could we not use a "oldest committed or rolled back" level instead ?. Maybe it has something to do with bitwise testing of the 2 bit states. Or the OIT is used by the sweep: garbage can only start for record versions with transactions id greater than the OIT.

In any case this OIT value is maintained and used to check whether sweep should be performed.

- in many documents the OIT is also used as starting id of the snapshot's private transaction list. In our simulator, we used the "oldest active". So we must have missed something there.

5.5 - Other transaction states

5.5.1 - Limbo transactions

There is a "limbo" state, which only occurs when connections to several databases are used. To commit or rollback those, a two phase mechanism is used. Due to some failure (network outage) this process is stopped before completion, leaving the transaction in this limbo state.

If in one of the databases a record version with a limbo transaction is found, we cannot garbage collect it outright, since the two phase commit or rollback could still be going on.

So we have to use an external tool GFIX to remove those.

Limbo transactions freeze the OIT. And since the OIT is the starting id of the private transaction lists, this will lead to long private transaction lists.

Limbo transactions cannot be removed by sweep, and if the sweep threshold has been reached, limbo will continue to trigger sweeps.

We did not consider the case of many databases in our simulator.

5.5.2 - Dead transactions

And some transaction might be "dead". Due to some failure, it can no longer be used. This is managed by the lock manager:

- each "live" transaction has a lock on its own id and a shared lock on the whole database.
- when a transaction T1 starts
 - it tries to get an exclusive lock on the database. If this succeeds, then there are no other transactions "live". So any other transaction in the active state is switched to rolled back.
 - the transaction then gets a lock on its own id.
- when later another transaction T2 tries to update or delete a record A, and the most recent record version A was created by T1, and T1 is active, T2 checks that T1 is still "live" (it tries to place a lock on T1's id. It should fail. If it succeeds, T1 is "dead"). If T1 is "dead", its state is switched from active to rolled back

This detection and removal of dead transactions is automatic.

5.6 - Deadlocks

When a modification occurs on a row modified by another transaction, when using the "no wait" mode, Interbase / Firebird trigger a Delphi exception (script 40):

```

01 START T1
02 c T1 A 800
03 COMM T1
04  START T2
05  u T2 A 801
06    START T3
07    u T3 A 802 *** lock_ver 102

```

and the exception message is (Delphi 6, Firebird 2.5, lbx) :

```

lock conflict on no wait transaction
deadlock
update conflicts with concurrent update
concurrent transaction number is 6

```

And:

- T2 places a lock
- T3 cannot update because of this lock

However we would call this a "deadlock" if T2 locks T3 and T3 locks T2 (deadly embrace). This is not the case here.

Here two transactions (applications) place locks on 2 different keys (script 41):

```

01 START T1
02 c T1 A 800
03 c T1 B 950
04 COMM T1
05  START T2
06  u T2 A 801
07    START T3
08    u T3 B 955
09    u T3 A 802 *** lock_ver 103
10  u T2 B 999 *** lock_ver 104

T1  commit
T2  active
T3  active
-- oat=T2 oast=- oatoast=-

101 A  800 (T1  commit)
102 B  950 (T1  commit)
103 A  801 (T2  active) x [ -> 101]
104 B  955 (T3  active) x [ -> 102]

```

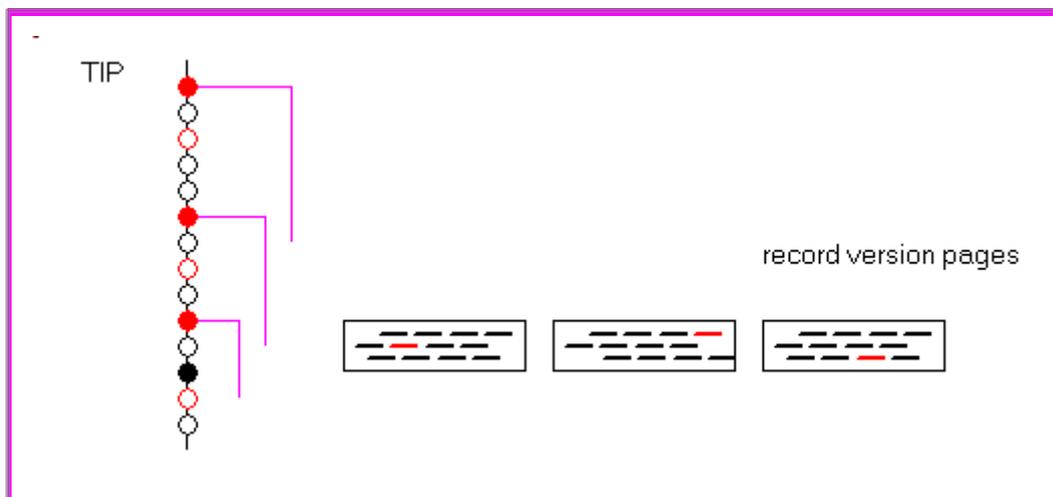
and in wait mode, the two applications would be stalled (blocked on the update call).

6 - Interbase / Firebird transaction optimization

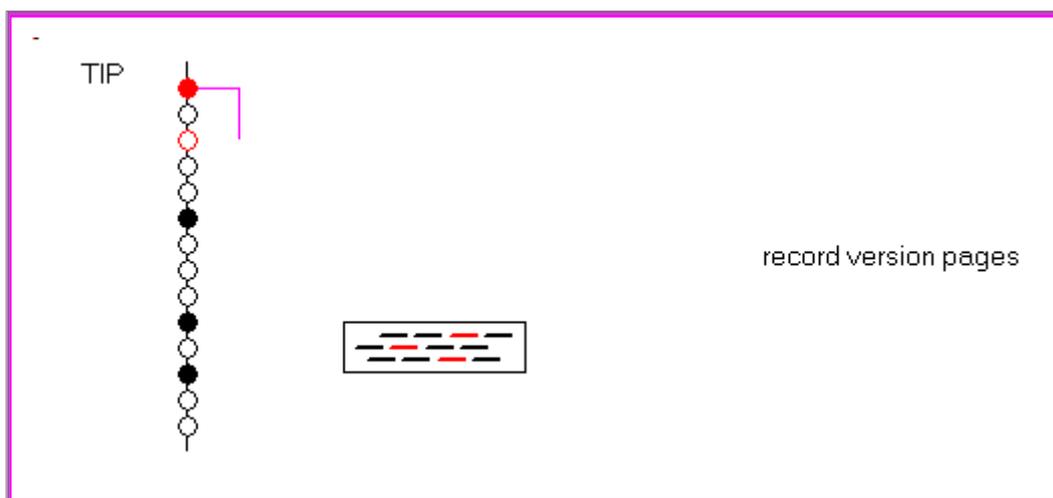
6.1 - Many non garbage collected record versions

Aside from sheer computation time (stored procedure processing large row volumes with complex queries) the main factor impacting speed is the count of non garbage collected record versions.

In a very naive way, the database would look like this



instead of that :



We will outline below a couple of hints about how we could improve the processing time caused by non garbage collected rows.

6.2 - Limbo transactions

If we do not use transactions spanning several databases, they should not occur.

We read a forum thread talking about a "phantom connection" caused by wrong connection parameters (the IP part was missing) but we did not try to reproduce it.

In any case the diagnostic is simple: "GFIX -list" will display the limbo transaction, and GFIX -commit or GFIX -rollback will remove those limbos.

So from now on, limbos are out of the picture.

6.3 - Snapshot transactions

Snapshot transactions offer repeatable reads. From this point of view they should be the preferred isolation mode. In fact, it is the Interbase / Firebird default isolation mode.

So read committed should be used only when we are adamant about reading the last committed version. We do not want to read the version that was committed when our transaction started, but the version committed before our read.

However snapshots create those private transaction lists. And if we have very old active transactions, those lists could become very long.

To solve this we could limit

- either the use of snapshots
- or the long transactions.

From our point of view, snapshots are desirable only if the application requires repeatable reads. If you only write values, with some updates, and read the values back (but without the constraint of a fully consistent picture of the database), read committed could be used.

Carefull use of shapshots would then reduce the burden of storing and sifting thru those long private transaction lists

The other solution is the careful management of the long transactions

6.4 - Long transactions

6.4.1 - Why long transactions

As a general rule, transactions should be as short lived as possible. Committing it allows other transaction to see our modification, and rolling them back allows garbage collection.

Some applications still could benefit from long transactions, like monitoring applications, where some action has to be performed every 10 ou 20 seconds. Using a transaction at each time could eat up lots of transaction ids', and since this is a 32 bit signed number, after 2 billion transactions, the server would require a backuprestore. In this case, using a single active transaction could be useful.

6.4.2 - The Delphi way

Delphi lead us to drop a *tQuery* on a *Form*, along with associated *tDataSource* and *tdbGrid*, and keep those open during the lifetime of our application. So this often led to "long transactions".

To avoid this, often committing or rolling back a transaction would cut the lifetime of the transaction down. But then all *tDataSet* linked to this transaction will be closed.

So commit-retaining and rollback-retaining were implemented.

6.4.3 - Commit-retaining and Rollback-retaining

Commit-retaining "Commits a transaction's changes, and preserves the transaction context for further transaction processing"

As we understand it, commit-retaining

- switches the state to committed
- instead of closing the cursor (+- the *tDataSet* buffers) completely, it causes a new transaction to be started retaining the cursor (= keeping the *tDataSets* open)
- if a snapshot is started after this commit-retaining, its "oldest active when this snapshot started" value will be the new started transaction. So the size of the private transaction lists of those new snapshot is lower when commit-retaining is peformed rather than keeping the transaction active.
- but the snapshots started before this commit-retaining will still keep their "oldest active when they started" value, since they still must be able to produce a repeatable read of whatever was active when they started. So no garbage processing can be performed on those row versions.
- if the snapshot itself does a commit-retaining this also has no influence on the rows

which must be kept around, for the same reasons (repeatable read), even if the snapshot was its own "oldest active" when it started.

So the retaining flavor

- keeps the *tDataset* open
- allows other to see our modifications earlier
- reduces the size of the private transaction lists of snapshots started after this retaining
- has no influence on garbage collection if snapshots are involved.

6.4.4 - Garbage collection Yield

If many record versions are generated (INSERTs or DELETES), it is possible that the garbage collector has not the time to keep up removing them. The SWEEP_YIELD_TIME (adjustable using IbConfig) can be set to 0, which will give the garbage collector thread the same priority as the user threads.

6.4.5 - Conclusion

Before the "immediate removal if less than 100.000 modification" rule, long transactions caused rolled back record versions to gradually accumulate until a sweep occurred.

Since this "immediate removal" is implemented, this is less frequent. In this case, for long running transactions :

- read committed read only transactions have no impact on the performance
- read committed read write transaction
 - regular commit-retaining will keep the performance impact low
 - rollback retaining with less than 100.000 modifications will be somehow impacted by the removal of the modified rows. With more than 100.000 modifications, the performance hit will be the slowdown of the garbage
- snapshot transactions : the commit-retaining or rollback-retaining do not systematically avoid performance degradation. Hard commit or hard rollback should

In any case if Interbase / Firebird seems to slow down, the solution is to use GSTAT and see if OST-OIT is large. If this is the case, many record versions have not been garbage collected. Sweep can temporarily help. But the solution is to see which transaction froze the OIT and fix this problem.

7 - Improvements

Basically our code should be seriously refactored

- removing duplicate methods
- consolidating tests and checks before the basic CRUD operations
- adding a level to our "managers", to allow separate execution of Firebird (currently Firebird is run after our simulator, and the simulator manages the simulation options).

8 - Download the Delphi Source code

Here are the source code files:

- [firebird_transaction_simulator.zip](#): the full project (116 K) :
 - the units
 - the scripts

The zip should be path independent. If the download of the .ZIP produces a .EXE, change this extension in .ZIP (bug of our current HTTP server)

As usual:

- please tell us at fcolibri@felix-colibri.com if you **found some errors, mistakes, bugs, broken links or had some problem downloading the file**. Resulting corrections will be helpful for other readers
- we welcome any **comment, criticism, enhancement, other sources or reference suggestion**. Just send an e-mail to fcolibri@felix-colibri.com.
- or more simply, enter your (anonymous or with your e-mail if you want an answer) comments below and **clie the "send" button**

Name :	<input type="text"/>
E-mail :	<input type="text"/>
Comments * :	<input type="text"/>

- and if you liked this article, **talk about this site** to your fellow developpers, **add a link to your links page** ou **mention our articles in your blog or newsgroup posts** when relevant. That's the way we operate: the more traffic and Google references we

get, the more articles we will write.

9 - Links and References

The absolute reference on the inner working of Interbase / Firebird is Ann HARRISON. Among her many papers and forum threads:

- [Firebird for the database expert: episode 4 - OAT, OIT and sweep](#)
Ann Harrison - there is no date, but our download is dated October 2005 the IbPhoenix site is down at the time of this writing, but google should bring other copies of this article

On transaction, especially with Delphi:

- [Understanding Transaction Lifetimes](#)
Craig STUNTZ - 2004 06 22
- [Understanding InterBase Transactions](#)
Bill TODD - October 2005 - Borcon 2004

Our site also contains several Interbase / Firebird articles:

- [Interbase Stored Procedure Grammar](#) : The BNF Grammar of the Interbase Stored Procedure. This grammar can be used to build stored procedure utilities, like pretty printers, renaming tools, Sql Engine conversion or ports
- [Using InterBase System Tables](#) :The Interbase / FireBird System Tables: description of the main Tables, with their relationship and presents examples of how to extract information from the schema
- and in French:
 - *Interbase*
 - [Interbase Tutorial](#) : programmation Client Serveur *Interbase* en Delpi
 - [Interbase dbExpress](#) : programmation de base de donnée *Interbase* en mode *dbExpress*
 - [Interbase lbx .Net](#): programmation de base de donnée *Interbase* en mode *lbx* avec *Delphi 8* et *Vcl.Net*
 - [Interbase Blobs](#) : gestion des Blobs Interbase pour stocker des données binaires, des textes ASCII et des images BMP ou JPEG
 - Firebird:
 - [Firebird Installation](#): installation du *Serveur* et du *Client* Firebird. Le détail pas à pas avec les vérifications à chaque étape
 - [Firebird Ado Net Tutorial](#): *ADO .Net* Tutorial, utilisant *Firebird*. Une introduction complète au développement *ADO .Net*, utilisant

SqlConnection, SqlCommand, SqlDataAdapter pour se connecter à un *Serveur*, exécuter directement du *SQL*, travailler avec des tables en mémoire, utiliser des *DataGrids* pour afficher et modifier les données. Très nombreux schémas et code source complets

10 - The author

Felix John COLIBRI works at the **Pascal Institute**. Starting with Pascal in 1979, he then became involved with Object Oriented Programming, Delphi, Sql, Tcp/Ip, Html, UML. Currently, he is mainly active in the area of **custom software development** (new projects, maintenance, audits, BDE migration, Delphi Xe_n migrations, refactoring), **Delphi Consulting** and **Delph training**, and is a frequent speaker at Borland Developer Conferences. His **web site** features tutorials, technical papers about programming with full downloadable **source code**, and the **description and calendar** of forthcoming **Delphi, Interbase, Asp.Net, Ado.Net and OOP / UML** training sessions.

Created: jul-13. Last updated: jul-13 - 98 articles, 131 .ZIP sources, 1012 figures

Copyright © Felix J. Colibri <http://www.felix-colibri.com> 2004 - 2013. All rights reserved

Link:

http://www.felix-colibri.com/papers/db/firebird_transaction_simulator/firebird_transaction_simulator.html#firebird_transaction_simulator