# Firebird for the database expert: episode 4 - OAT, OIT and sweep

By Ann Harrison

This is an ancient message from an InterBase® self-help list, responding to a question about slow inserts. It deals with questions of sweeping, oldest active transaction, oldest interesting transaction, etc. I've cleaned up the spelling and added a few side notes.

**From:** Ann Harrison

**Subject:** Re: Interbase® - what is it doing?

Let me also take a crack at this, since I may be the only person with more experience trying to explain it than Jim (Starkey - my previous & current boss/mentor/(he says "say husband") etc.). The problem may be a sweep.

First, for Novice InterBasians (and fresh-hatched Firebirdies) - when I say transaction, I mean a set of actions against the database, ending with a Commit, Rollback, Prepare/Commit (two-phase commit), or abrupt disconnection from the database. A single action, like inserting, updating, or deleting a record is a statement. Many tools provide automatic transaction support, so you may not be aware of the number of transactions created on your behalf. Any tool that performs a commit per statement is not your friend if you're loading a database.

Here's the hard-core stuff.

Explanations of sweeping tend to be unsatisfactory because the subject is complicated, and depends on understanding several other complicated ideas.

**Disclaimer:** This description applies to the state of the world in V3.x, with extrapolation to V4.x specifically noted. I have no current connection with InterBase® or Borland. (See note 1 in the Summary).

Lets begin by defining transaction states, garbage, garbage collection, and Oldest Interesting Transaction (OIT), Oldest Active Transaction, and sweeping...

### Transaction states

Transactions have four states: active, committed, limbo, and rolled back.

Taking these cases in order from the least complex to the most:

- **Limbo:** A transaction that started a two-phase commit by calling the PREPARE routine. The transaction may be alive or not. At any point, the transaction may re-appear and ask to COMMIT or ROLLBACK. Changes it made can neither be trusted nor ignored, and certainly cannot be removed from the database.
- **Committed:** A transaction is which completed its activity successfully. Either A) it called COMMIT and the commit completed successfully, or B) it called ROLLBACK but made no

changes to the database, or C) it called ROLLBACK and its changes were subsequently undone and its state changed to committed. This transaction is finished and will never be heard from again, and its remaining changes are now officially part of the database.
- **Rolled back:** A transaction which either: A) called ROLLBACK and requested that its changes be removed from the database, or B) never called COMMIT so was marked as ACTIVE, but discovered to be dead by another transaction which marked it as rolled back. In either case, changes made by this transaction must be ignored and should be removed from the database.
- **Active:** A transaction which: A) hasn't started. B) has started and hasn't finished. C) started and ended without calling any termination routine. (e.g. crashed, lost communication, etc.)

**How do transactions know about each others state?**

The state of every transaction is kept on a Transaction Inventory Page (TIP). The single change made to the database when a transaction commits is to change the state of the transaction from ACTIVE to COMMITTED. When a transaction calls the rollback routine, it checks its *Update* flag - if the flag is not set, meaning that no updates have been made, it calls COMMIT instead. So, rolling back read-only transactions doesn't mess up the database.

**How can a transaction go back from Active to Rolled Back if it exists abnormally?**

This can happen in one of two ways:

1. When a transaction starts, it takes out a lock on its own transaction id. If a transaction (B) attempts to update or delete a record and finds that the most recent version of the record was created by a transaction (A) whose TIP state is ACTIVE, transaction B tries to get a conflicting lock on A's transaction id. A live transaction maintains an exclusive lock on its own id, and the lock manager can probe a lock to see if the owner is still alive. If the lock is granted, then B knows that A died and changes A's TIP state from ACTIVE to ROLLED BACK.

2. When a transaction starts, it checks to see if it can get an exclusive lock on the database - if it can no other transactions are active. Every active transaction has a shared lock on the database. If it gets an exclusive lock, it converts all Active TIP entries to ROLLED BACK.

To reiterate, a transaction is ACTIVE (meaning that it appears to be alive), LIMBO (meaning that its outcome can not be determined), COMMITTED (meaning that it completed successfully) or ROLLED BACK (meaning it acknowledged its faults and left the field in disgrace).

back to top of page

**Garbage**

InterBase® is a multi-generational database. When a record is updated, a copy of the new values is placed in the database, but the old values remain (usually as a bytewise difference from the new value). The old value is called a “Back Version”. The back version is the rollback log - if the transaction that updated the record rolls back, the old version is right there, ready to resume its old place. The back version is also the shadow that provides repeatable reads for long running transactions. The version numbers define which record versions particular tranasctions can see.

When the transaction that updated the record commits and all concurrent transactions finish, the back version is unnecessary. In a database in which records are updated significantly and regularly, unnecessary back versions could eventually take up enough disk space that they would reduce the

performance of the database. Thus they are GARBAGE, and should be cleaned out.

### Garbage collection

Garbage collection prevents an update-intensive database from filling up with unnecessary back versions of records. It also removes record versions created by transactions that rolled back. Every transaction participates in garbage collection - every transaction, including read-only transactions.

When a client application reads a record from a Firebird database, it gets a record that looks like any record from any database. Two levels lower, somewhere in the server, Firebird/InterBase® pulls a string of record versions off the disk. Each version is tagged with the transaction id of the transaction that created it. The first one is the most recently stored. At this point, the server has two goals: 1) produce an appropriate version of the record for the current transaction 2) remove any versions that are garbage - either because they were created by a transaction that rolled back or because they are so old that nobody will ever want to see them again.

**Extra Credit Aside:** There is a third kind of garbage collection which happens at the same time. InterBase® also uses a “multi-generational” delete. When transaction deletes a record, does the record go away right then? No, of course not. The deletion could be rolled back. So instead of removing the record, InterBase® sticks in a new record version containing only a DELETE marker, and keeps the old version. Sooner or later the deletion commits and matures. Then the whole thing, deletion marker and all record versions are garbage and get … (right you are!) garbage collected.

### Garbage Collection – resumes:

Garbage collection is co-operative, meaning that all transactions participate in it, rather than a dedicated garbage team. Old versions, deleted records, and rolled back updates are removed when a transaction attempts to read the record. In a database where all records are continually active, or where exhaustive retrievals (i.e. non-indexed access) are done regularly on all tables, co-operative garbage collection works well, as long as the transaction mask stays current.

For databases in which all access is indexed, old records are seldom - or never - revisited and so they seldom - or never - get garbage collected. Running a periodic backup with gbak has the secondary effect of forcing garbage collection since gbak performs exhaustive retrievals on all tables.

### See also:

- Garbage collection
- Garbage collectors
- Database housekeeping and garbage collection
- Firebird administration using IBExpert: Garbage collection
- How do you know if your database server garbage collection is working?
- Firebird 2.1.3 Release Notes: Garbage collector rationalisation

back to top of page

### Oldest Interesting Transaction (OIT)

To recognize which record versions can garbage collected, and which updates are rolled back and can

be ignored, every transaction includes a transaction mask which records the states of all interesting transactions. A transaction is interesting to another transaction if it is concurrent - meaning that its updates are not committed, or if it rolled back - meaning that its updates should be discarded, or if it's in limbo.

The transaction mask is a snapshot of the states of all transactions from the oldest interesting, to the current. The snapshot is made when the transaction starts and is never updated. The snapshot depends on the number of transactions that have started since the oldest interesting transaction.

### Oldest Active Transaction (OAT)

This one sounds easy - but it's not. The oldest active transaction is not the oldest transaction currently running. Nor is it the oldest transaction marked ACTIVE in the TIP. (Alas). It is the oldest transaction that was active when the oldest transaction currently active started. The bookkeeping on this is hairy and I frankly don't remember how it was done - now I do -, but that's the rule, and it does work.

Any record version behind a committed version created by a transaction older than the oldest transaction active when the oldest transaction currently active started is garbage and will never be needed ever again.

That's pretty dense. Lets ignore the commit/rollback question briefly.

**Simple case:** I'm transaction 20 and I'm the only transaction running. I find a record created and committed by transaction 15. I modify it and commit. You are transaction 25, and when you start, you are also the only transaction active. You read the same record, recognize that all active transactions can use the version of the record created by me, so you garbage collect the original version. In this case, your threshold for garbage collection (aka Oldest Active) is yourself.

**Harder case:** You continue puttering around, modifying this and that. Another transaction, say 27 starts. You are its oldest active. It too can modify this and that, as long as it doesn't modify anything you modified. It commits. I start a transaction 30. You are also my oldest active transaction, and I can't garbage collect any record version unless the newer version is older than you. I run into a record originally created by transaction 15, modified by transaction 20, then modified again by 27. All three of those transactions are committed, but I can garbage collect only the original version, created by transaction 15. Although the version created by transaction 27 is old enough for me, it is not old enough for you, and being cooperative, I have to consider your needs too.

**Hardest case:** I'm transaction 87, and when I started, all transactions before 75 had committed, and everybody from 75 on was active. Transaction 77 modifies a record, created originally by transaction 56. I continue to read the 56 version. All is well. Transaction 77 commits. You are transaction 95. When you start, I, number 87, am the oldest active. You read the record created by 56 and modified by 77. You can't garbage collect anything in that record because I can't read records created by any transaction newer than 74.

Maybe you know now why descriptions of the oldest active tend to be a little peculiar.

back to top of page

### Sweeping

Sweeping is NOT just organized garbage collection. What sweeping seeks to do is to move the Oldest Interesting Transaction up, and reduce the size of transaction masks. It does so by changing rolled back transactions to committed transactions.

“What!!!”, you say. “The woman is nuts.”

But that's what a sweep does. It removes all the changes made by a rolled back transaction then changes it state to committed. (Remember we agreed earlier that a read-only transaction that rolled back could be considered committed for all the harm it did. Remove the damage, and its safe to consider the transaction committed.)

At the same time, sweep garbage collects like any other transaction.

Prior to version 4.2, the unlucky transaction that triggered the sweep gets to do the work. Other concurrent transactions continue, largely unaffected. In version 4.2 and later, a new thread is started and sweeps the database while everybody else goes about life as normal. Well, more or less normal, where the less is the amount of CPU and I/O bandwidth used by the sweep.

**See also:**

- Database sweep / sweep interval
- Database repair and sweeping using GFIX

### Aside on limbo transactions

A transaction in limbo cannot be resolved by a sweep, will continue to trigger sweeps, and will block attempts to update or delete record versions it created. However, InterBase® gives good diagnostics when it encounters a record in that state, and no tool is likely to generate incomplete two-phase commits on a random basis.

back to top of page

# Some examples

The unfortunate case that started this message was an attempt to insert 1,000,000 records, one transaction, and one commit per record. The process slowed to a crawl, which was blamed on sweeps. sweeping may be the problem, but I doubt it.

### Case 1

Single stream of non-concurrent transactions. Transaction 1 inserts record 1, and commits. Transaction 2 starts and is both oldest active and oldest interesting. It inserts record 2 and commits. Transaction 3 starts, is oldest active and oldest interesting, inserts its record and commits. Eventually, transaction 1,000,000 starts and it too is both oldest interesting and oldest active. No sweeps.

### Case 2

Lurker in the background. Transaction 1 starts, looks around, and goes off for a smoke. Transaction 2 starts, notices that 1 is oldest interesting and oldest active, inserts record 1 and commits. Transaction 3 starts, notices that 1 is still OI and OA, inserts record 2 and commits. Eventually transaction 1,000,001 starts, notices that 1 is still OI and OA so the difference between the two is still 0, stores, and commits. No sweeps again.

### Case 3

*Suicidal lurker*. Transaction 1 starts, does something, goes out for a smoke. Transaction 2 starts, notices that 1 is oldest interesting and oldest active, inserts record 1 and commits. Transaction 3 starts, notices that 1 is still OI and OA, inserts record 2 and commits. Eventually transaction 1 succumbs to smoke inhalation and dies quietly in his corner. Transaction 15,034 (by luck) starts, gets an exclusive lock on the database, and sets Transaction 1's state to Rolled Back. Now the oldest interesting is still 1, but the oldest active is 15,034. The difference is 15,033, so no sweep yet. 4,967 transactions later the sweep occurs. Depending on the version of InterBase®, transaction 20,001 may actually be charged with the time spent sweeping. Versions since 4.1 start a new thread. Once the sweep is done, the OI and OA march up together, hand in hand, and there is no more sweeping unless another transaction goes into an interesting and non-active state.

### Case 4

Suicidal Twin. If for every record stored, the tool started one transaction which stored the record then rolled back, followed by a second transaction which stored the record and committed, then the difference between the OA and the OI would go up one for each record successfully stored. (Transaction 1 becomes OI when it rolls back. Transaction 2 is OA when it starts and the difference is 1. Transaction 3 rolls back, but is not OI because Transaction 1 is still older. Transaction 4 is OA and sees a difference of 3 between it and Transaction 1, and so on until transaction 20,001 which sweeps, and brings the OA and OI together at 20,001. Unfortunately its only storing record 10,001 since half the attempts to store are failing. In this EXTREMELY UNLIKELY case, storing 1,000,000 records would cause 100 sweeps. However, it would require an UNUSUALLY bad programmer to create anything that AMAZINGLY inefficient. Grounds for a career change.

Summary Beats me why the load was so slow, although the commit per insert does a lot more writing than just inserting. That and forced write might explain a lot. Maybe a really fragmented disk?

**Note 1:** This message was written sometime last century, before I got involved with InterBase® and then Firebird. I now know a lot more about InterBase® 4.x, 5.x, 6.x and Firebird 1.0x, 1.5x, 2.0x, and Vulcan. That knowledge will show up passim.