The InterBase and Firebird Developer Magazine

#2

2005 | Issue 2

Locking, Firebird, and the Lock Table

- Oldest Active by Helen Borrie
- Inside BLOBs
- Testing NO SAVEPOINT in InterBase 7.5.1
- OOD and RDBMS, Part 1
- Using IBAnalyst

www.ibdeveloper.com

Contents

Magazine CD

Best viewed with Acrobat Reader 7
Download now!

Credits

Alexey Kovyazin,
Chief Editor

Helen Borrie,
Editor

Dmitri Kouzmenko
Editor

Noel Cosgrave,
Sub-editor

Lev Tashchilin,
Designer

Natalya Polyanskaya,
Blog editor

Subscribe now!

To receive future issues notifications send email to

subscribe@ibdeveloper.com

Donations

Rock around the blog

Editor's note

By Alexey Kovyazin

Dear readers,

I am glad to introduce you the second issue of "The InterBase and Firebird Developer Magazine". We received a lot of emails with kind words and congratulations which have helped us in creating this issue. Thank you very much!

In this issue

The cover story is the newest Episode from Ann W. Harrison "Locking, Firebird, and the Lock Table". The article covers the locking issue from general principles to specific advice, so everyone will find a bit that's interesting or informative.

We continue the topic of savepoints internals with an article "Testing NO SAVEPOINT in InterBase 7.5.1". Along with interesting practical test results you will find a description of UNDO log workings in InterBase 7.5.1.

We publish the small chapter, "Inside BLOBs" from the forthcoming book "1000 InterBase&Firebird Tips&Tricks" by Dmitri Kouzmenko and Alexey Kovyazin.

Object oriented development has been a high-interest topic for many years, and it is a still hot topic. The article "Object-Oriented Development in RDBMS" explores the practical use of OOD principles in the design of InterBase or Firebird databases.

Replication is a problem which faces every database developer sooner or later. The article "Replicating and synchronizing InterBase/Firebird databases using CopyCat" introduces the approach used in the new CopyCat component set and the CopyTiger tool.

The last article by Dmitri Kouzmenko "Understanding Your Database with IBAnalyst" provides a guide for better understanding of how databases work and how tuning their parameters can optimize performance and avoid bottlenecks.

"Oldest Active" column

I am very glad to introduce the new column "Oldest Active" by Helen Borrie. No need to say more about author of "The Firebird Book", just read it! In this issue Helen looks at the phenomenon of support lists and their value to the Firebird community. She takes a swing at some of the unproductive things that list posters do in this topic, titled "On Silly Questions and Idiotic Outcomes".

Let's blog again

Now it is a good time to say a few words about our new web-presentation. We've started a blog-style interface for our magazine – take a look on www.ibdeveloper.com if you haven't already discovered it.. Now you can make comments on any article or message. In future we'll publish all the materials related to the PDF issues, along with special bonus articles and materials. You can look out for previews of articles, drafts and behind-the-scene community discussions. Please feel welcome to blog with us!

Donations

The magazine is free for readers, but it is a considerable expense for its publisher. We must pay for articles, editing and proofreading, design, hosting, etc. In future we'll try to recover costs solely from advertising but, for these initial issues, we need your support.

See details here

http://ibdeveloper.com/donations

On the Radar

I'd like to introduce the several projects which will be launched in the near future. We really need your feedback so please do not hesitate to place your comments in blog or send email to us (readers@ibdeveloper.com)

Magazine CD

We will issue a CD with all 3 issues of "The InterBase and Firebird Developer Magazine" in December 2005 (yes, a Christmas CD!).

Along with all issues in PDF and searchable html-format you will find exclusive articles and bonus materials. The CD will also include free versions of popular software relat-

coming soon

ed to InterBase and Firebird and offer very attractive discounts for dozens of the most popular products.

The price for CD is USD$9.99 plus shipping.

For details see
www.ibdeveloper.com/magazine-cd

Paper version of the magazine

In 2006 we intend to issue the first paper version of "The InterBase and Firebird Developer Magazine". The paper version cannot be free because of high production costs.

We intend to publish 4 issues per year with a subscription price of around USD$49. The issue volume will be approximately 64 pages.

In the paper version we plan to include the best articles from the online issues and, of course, exclusive articles and materials.

To be or not be – this is the question that only you can answer. If the idea of subscribing to the paper version appeals to you, please place a comment in our blog (http://ibdeveloper.com/paper-version) or send email to readers@ibdeveloper.com with your pro and cons. We look forward for your feedback!

Invitation

As a footnote, I'd like to invite all people who are in touch with Firebird and InterBase to participate in the life of the community. The importance of a large, active community to any public project -- and a DBMS with hundreds of thousands of users is certainly public! -- cannot be emphasised enough. It is the key to survival for such projects. Read our magazine, ask your questions on forums, and leave comments in blog, just rock and roll with us!

Sincerely,

Alexey Kovyazin

Chief Editor

editor@ibdeveloper.com

Firebird Conference

This issue of our magazine almost coincides with the third annual Firebird World Conference, starting November 13 in Prague, Czech Republic. This year's conference spans three nights, with a tight programme of back-to-back and parallel presentations over two days. With speakers in attendance from all across Europe and the Americas presenting topics that range from specialised application development techniques to first-hand accounts of Firebird internals from the core developers, this one promises to be the best ever.

Registration has been well under way for some weeks now, although the time is now past for early bird discounts on registration fees. At the time of publishing, bookings were still being taken for accommodation in the conference venue itself, Hotel Olsanka, in downtown Prague. The hotel offers a variety of room configurations, including bed and breakfast if wanted, at modest tariffs. Registration includes lunches on both conference days as well as mid-moring and mid-afternoon refreshments.

Presentations include one on Firebird's future development from the Firebird Project Coordinator, Dmitry Yemanov and another from Alex Peshkov, the architect of the security changes coming in Firebird 2. Jim Starkey will be there, talking about Firebird Vulcan, and members of the SAS Institute team will talk about aspects of SAS's taking on Firebird Vulcan as the back-end to its illustrious statistical software. Most of the interface development tools are on the menu, including Oracle-mode Firebird, PHP, Delphi, Python and Java (Jaybird).

It is a "don't miss" for any Firebird developer who can make it to Prague. Link to details and programme either at http://firebird.sourceforge.net/index.php?op=konferenz or at the IBPhoenix website, http://www.ibphoenix.com.

On Silly Questions and Idiotic Outcomes

Firebird owes its existence to the community's support lists. Though it's all ancient history now, if it hadn't been for the esprit de corps among the "regulars" of the old InterBase support list hosted by mers.com in the 'nineties, Borland's shock-horror decision to kill off InterBase development at the end of 1999 would have been the end of it, for the English-speaking users at least.

Firebird's support lists on Yahoo and Sourceforge were born out of the same cataclysm of fatal fire and regeneration that caused the phoenix, drawn from the fire against extreme adversity, to take flight as Firebird in July 2000. Existing and new users, both of Firebird and of the rich selection of drivers and tools that spun off from and for it, depend heavily on the lists to get up to speed with the evolution of all of these software products.

Newbies often observe that there is just nothing like our lists for any other software, be it open source or not. One of the things that I think makes the Firebird lists stand out from the crowd is that, once gripped by the power of the software, our users never leave the lists. Instead, they stick around, learn hard and stay ready and willing to impart to others what they have learnt. After nearly six years of this, across a dozen lists, our concentrated mix of expertise and fidelity is hard to beat.

Now, doesn't all this make us all feel warm and fuzzy? For those of us in the hot core of this voluntary support infrastructure, the answer has to be, unfortunately, "Not all the time!" There are some bugbears that can make the most patient person see red. I'm using this column to draw attention to some of the worst.

First and worst is silly questions. You have all seen them. Perhaps you even posted them! "Subject: Help! Body: Every time I try to connect to Firebird I get the message 'Connection refused'. What am I doing wrong?" What follows from that is a frustrating and tedious game for responders. "What version of Firebird are you using? Which server model? Which platform? Which tool? What connection path? ('By connection path, we mean....').."

Everyone's time is valuable. Nobody has the luxury of being able to sit around all day playing this game. If we weren't willing to help, we wouldn't be there. But you make it hard, or even impossible, when you post problems with no descriptions. You waste our time. You waste your time. We get frustrated because you don't provide the facts; you get frustrated because we can't read your mind. And the list gets filled with noise.

If there's something I've learnt in more than 12 years of on-line software support, it is that a problem well described is a problem solved. If a person applies time and thought to presenting a good description of the problem, chances are the solution will hop up and punch him on the nose. Even if it doesn't, good descriptions produce the fastest right answers from list contributors. Furthermore, those good descriptions and their solutions form a powerful resource in the archives, for others coming along behind.

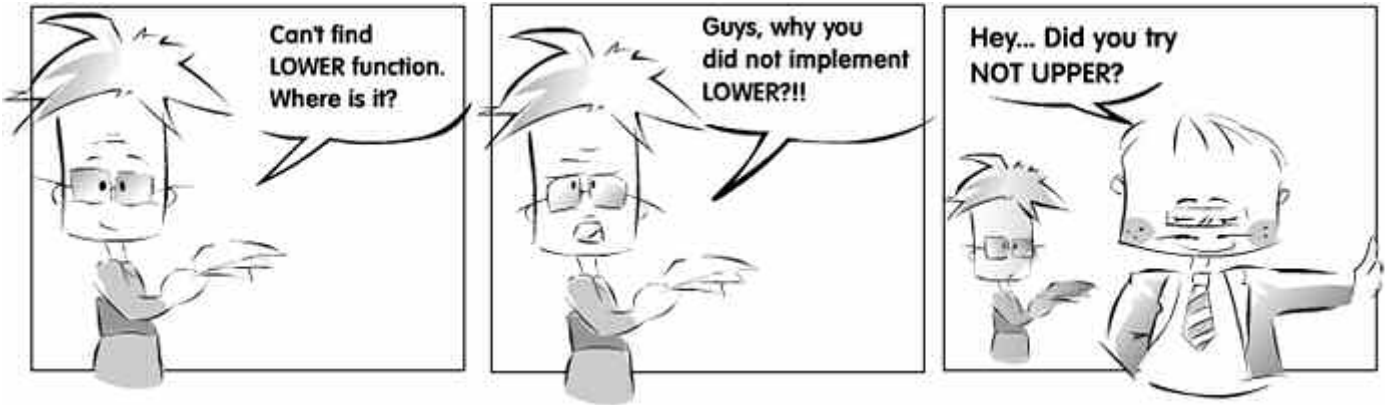
Off-topic questions are another source of noise and irritation in the lists. At the Firebird website, we have bent over backwards to make it very clear which lists are appropriate for which areas of interest. An off-topic posting is easily forgiven if the poster politely complies with a request from a list regular to move the question to "x" list. When these events instead become flame threads, or when the same people persistently reoffend, they are an extreme burden on everyone.

In the "highly tedious" category we have the kind of question that goes like this: "Subject: Bug in Firebird SQL. Body: "This statement works in MSSQL/Access/MySQL/insert any non-standard DBMS name here. It doesn't work in Firebird. Where should I report this bug?" To be fair, George Bernard Shaw wasn't totally right when he said "Ignorance is a sin." However, you do at least owe it to yourself to know what you are talking about, and not to present your problem as an attack on our software. Whether intended or not, it comes over as a troll and you come across as a fool. There's a strong chance that your attitude will discourage anyone from bothering with you at all. It's no-win, sure, but it's also a reflection of human nature. You reap what you sow.

In closing this little tirade, I just want to say that I hope I've struck a chord with those of our readership who struggle to get satisfaction from their list postings. It will be a rare question indeed that has no answer. Those rarities, well-presented, become excellent bug reports. If you're not getting an answer, there's a high chance that you asked a silly question.

Helen E. M. Borrie

helebor@tpg.com.au



News & Events

Firebird Worldwide Conference 2005

The Firebird conference will take place at the Hotel Olsanka in Prague, Czech Republic from the evening of Sunday the 13th of November (opening session) until the evening of Tuesday the 15th of November (closing session).

Request for Sponsors

How to Register

Conference Timetable

Conference Papers and Speaker

Locking, Firebird, and the Lock Table

Author: Ann. W. Harrison
aharrison@ibphoenix.com

One of the best-known facts about Firebird is that it uses multi-version concurrency control instead of record locking. For databases that manage concurrency though record locks, understanding mechanisms of record locking is critical to designing a high performance, high concurrency application. In general, record locking is irrelevant to Firebird. However, Firebird uses locks to maintain internal consistency and for interprocess coordination. Understanding how Firebird does (and does not) use locks helps with system tuning and anticipating how Firebird will behave under load.

I'm going to start by describing locking abstractly, then Firebird locking more specifically, then the controls you have over the Firebird lock table and how they can affect your database performance. So, if you just want to learn about tuning, skip to the end of the article.

Concurrency control

Concurrency control in database systems has three basic functions: preventing concurrent transactions from overwriting each others' changes, preventing readers from seeing uncommitted changes, and giving a consistent view of the database to running transactions. Those three functions can be implemented in various ways. The simplest is to serialize transactions – allowing each transaction exclusive access to the database until it finishes. That solution is neither interesting nor desirable.

A common solution is to allow each transaction to lock the data it uses, keeping other transactions from reading data it changes and from changing data it reads. Modern databases generally lock records. Firebird provides concurrency control without record locks by keeping multiple versions of records, each marked with the identifier of the transaction that created it. Concurrent transactions don't overwrite each other's changes, because the system will not allow a transaction to change a record if the most recent version was created by a concurrent transaction. Readers never see uncommitted data because the system will not return record versions created by concurrent transactions. Readers see a consistent version of the database because the system returns only the data committed when they start – allowing other transactions to create newer versions. Readers don't block writers.

A lock sounds like a very solid object, but in database systems, a lock anything but solid. "Locking" is a shorthand description of a protocol for reserving resources. Databases use locks to maintain consistency while allowing concurrent independent transactions to update distinct parts of the database. Each transaction reserves the resources – tables, records, data pages – that it needs to do its work. Typically, the reservation is made in memory to control a resource on disk, so the cost of reserving the resource is not significant compared with the cost of reading or changing the resource.

Locking example

"Resource" is a very abstract term. Lets start by talking about locking tables. Firebird does lock tables, but normally it locks them only to prevent catastrophes like having one transaction drop a table that another transaction is using.

Before a Firebird transaction can access a table, it must get a lock on the table. The lock prevents other transactions from dropping the table while it is in use. When a transaction gets a lock on a table, Firebird makes an entry in its table of locks, indicating the identity of the transaction that has the lock, the identity of the table being locked, and a function to call if another transaction wants an incompatible lock on the table. The normal table lock is shared – other transactions can lock the same table in the same way.

Before a transaction can delete a table, it must get an exclusive lock on the table. An exclusive lock is incompatible with other locks. If any transaction has a lock on the table, the request for an exclusive lock is denied, the drop table statement fails. Returning an immediate error is one way of dealing with conflicting lock requests. The other is to put the conflicting request on a list of unfulfilled requests and let it wait for the resource to become available.

In its simplest form, that is how locking works. All transactions follow the formal protocol of requesting a lock on a resource before using it. Firebird maintains a list of locked resources, a list of requests for locks on resources – satisfied or waiting – and a list of the owners of lock requests. When a transaction requests a lock that is incompatible with existing locks on a resource,

Firebird either denies the new request, or puts it on a list to wait until the resource is available. Internal lock requests specify whether they wait or receive an immediate error on a case-by-case basis. When a transaction starts, it specifies whether it will wait for locks that it acquires on tables, etc.

Lock modes

For concurrency and read committed transactions, Firebird locks tables for shared read or shared write. Either mode says, "I'm using this table, but you are free to use it too." Consistency mode transactions follow different rules. They lock tables for protected read or protected write. Those modes say "I'm using the table and no one else is allowed to change it until I'm done." Protected read is compatible with shared read and other protected read transactions. Protected write is only compatible with share read.

The important concept about lock modes is that locks are more subtle than mutexes – locks allow resource sharing, as well as protecting resources from incompatible use.

Two-phase locking vs. transient locking

The table locks that we've been describing follow a protocol known as two-phase locking, which is typical of locks taken by transactions in database systems. Databases that use record locking for consistency control always use two-phase record locks. In two-phase locking, a transaction acquires locks as it proceeds and holds the locks until it ends. Once it releases any lock, it can no longer acquire another. The two phases are lock acquisition and lock release. They cannot overlap.

When a Firebird transaction reads a table, it holds a lock on that table until it ends. When a concurrency transaction has acquired a shared write lock to update a table, no consistency mode transaction will be able to get a protected lock on that table until the transaction with the shared write lock ends and releases its locks. Table locking in Firebird is two-phase locking.

Locks can also be transient, taken and released as necessary during the running of a transaction. Firebird uses transient locking extensively to manage physical access to the database.

Firebird page locks

One major difference between Firebird and most other databases is Firebird's Classic mode. In Classic mode, many separate processes share write access to a single database file. Most databases have a single server process like SuperServer that has exclusive access to the database and coordinates physical access to the file within itself. Firebird coordinates physical access to the database through locks on database pages.

In general database theory, a trans-

action is a set of steps that transform the database from on consistent state to another. During that transformation, the resources held by the transaction must be protected from incompatible changes by other transactions. Two-phase locks are that protection.

In Firebird, internally, each time a transaction changes a page, it changes that page – and the physical structure of the database as a whole – from one consistent state to another. Before a transaction reads or writes a database page, it locks the page. When it finishes reading or writing, it can release the lock without compromising the physical consistency of the database file. Firebird page level locking is transient. Transactions acquire and release page locks throughout their existence. However, to prevent deadlocks, transactions must be able to release all the page locks it holds before acquiring a lock on a new page.

The Firebird lock table

When all access to a database is done in a single process – as is the case with most database systems – locks are held in the server's memory and the lock table is largely invisible. The server process extends or remaps the lock information as required. Firebird, however, manages its locks in a shared memory section. In SuperServer, only the server uses that share memory area. In Classic, every database connection maps the shared memory and every connection can read and change the contents of the memory.

The lock table is a separate piece of share memory. In SuperServer, the lock table is mapped into the server process. In Classic, each process maps the lock table. All databases on a server computer share the same lock table, except those running with the embedded server.

The Firebird lock manager

We often talk about the Firebird Lock Manager as if it were a separate process, but it isn't. The lock management code is part of the engine, just like the optimizer, parser, and expression evaluator. There is a formal interface to the lock management code, which is similar to the formal interface to the distributed lock manager that was part of VAX/VMS and one of the interfaces to the Distributed Lock Manager from IBM.

The lock manager is code in the engine. In Classic, each process has its own lock manager. When a Classic process requests or releases a lock, its lock management code acquires a mutex on the shared memory section and changes the state of the lock table to reflect its request.

Conflicting lock requests

When a request is made for a lock on a resource that is already locked in an incompatible mode, one of two things happens. Either the requesting transaction gets an immediate error, or the request is

put on a list of waiting requests and the transactions that hold conflicting locks on the resource are notified of the conflicting request. Part of every lock request is the address of a routine to call when the lock interferes with another request for a lock on the same object. Depending on the resource, the routine may cause the lock to be released or require the new request to wait.

Transient locks like the locks on database pages are released immediately. When a transaction requests a page lock and that page is already locked in an incompatible mode, the transaction or transactions that hold the lock are notified and must complete what they are doing and release their locks immediately. Two-phase locks like table locks are held until the transaction that owns the lock completes. When the conflicting lock is released, and the new lock is granted, then transaction that had been waiting can proceed.

Locks as interprocess communication

Lock management requires a high speed, completely reliable communication mechanism between transactions, including transactions in different processes. The actual mechanism varies from platform to platform, but for the database to work the mechanism must be fast and reliable. A fast, reliable interprocess communication mechanism can be – and is – useful for a number of purposes outside the area that's normally considered database locking.

For example, Firebird uses the lock table to notify running transactions of the existence of a new index on a table. That's important, since as soon as an index becomes active, every transaction must help maintain it – making new entries when it stores or modifies data, removing entries when it modifies or deletes data.

When a transaction first references a table, it gets a lock on the existence of indexes for the table. When another transaction wants to create a new index on that table, it must get an exclusive lock on the existence of indexes for the table. Its request conflicts with existing locks, and the owners of those locks are notified of the conflict. When those transactions are in a state where they can accept a new index, they release their locks, and immediately request new shared locks on the existence of indexes for the table. The transaction that wants to create the index gets its exclusive lock, creates the index, and commits, releasing its exclusive lock on the existence of indexing. As other transactions get their new locks, they check the index definitions for the table, find the new index definition, and begin maintaining the index.

Firebird locking summary

Although Firebird does not lock records, it uses locks extensively to isolate the effects of concurrent transactions. Locking and the lock table are more visible in Firebird than in other databases because the lock table is a central communication channel between the separate processes that access the database in Classic mode. In addition to controlling access to database objects like tables and data pages, the Firebird lock manager allows different transactions and processes to notify each other of changes to the state of the database, new indexes, etc.

Lock table specifics

The Firebird lock table is an in-memory data area that contains of four primary types of blocks. The lock header block describes the lock table as a whole and contains pointers to lists of other blocks and free blocks. Owner blocks describe the owners of lock requests – generally lock owners are transactions, connections, or the SuperServer. Request blocks describe the relationship between an owner and a lockable resource – whether the request is granted or pending, the mode of the request, etc. Lock blocks describe the resources being locked.

To request a lock, the owner finds the lock block, follows the linked list of requests for that lock, and adds its request at the end. If other owners must be notified of a conflicting request, they are located through the request blocks already in the list. Each owner block also has a list of its own requests. The performance critical part of locking is finding lock blocks. For that purpose, the lock table includes a hash table for access to lock blocks based on the name of the resource being locked.

A quick refresher on hashing

A hash table is an array with linked lists of duplicates and collisions hanging from it. The names of lockable objects are transformed by a function called the hash function into the offset of one of the elements of the array. When two names transform to the same offset, the result is a collision. When two locks have the same name, they are duplicates and always collide.

In the Firebird lock table, the array of the hash table contains the address of a hash block. Hash blocks contain the original name, a collision pointer, a duplicate pointer, and the address of the lock block that corresponds to the name. The collision pointer contains the

News & Events

PHP Server

One of the more interesting recent developments in information technology has been the rise of browser based applications, often referred to by the acronym "LAMP".

One key hurdle for broad use of the LAMP technology for mid-market solutions was that it was never easy to configure and manage.

PHPServer changes that: it can be installed with just four clicks of the mouse and support for Firebird is compiled in.

PHPServer shares all the qualities of Firebird: it is a capable, compact, easy to install and easy to manage solution.

PHPServer is a free download

Read more at:

www.fyracle.org/phpserver.html

address of a hash block whose name hashed to the same value. The duplicate pointer contains the address of a hash block that has exactly the same name.

A hash table is fast when there are relatively few collisions. With no collisions, finding a lock block involved hashing the name, indexing into the array, and reading the pointer from the first hash block. Each collision adds another pointer to follow and name to check. The ratio of the size of the array to the number of locks determines the number of collisions. Unfortunately, the width of the array cannot be adjusted dynamically because the size of the array is part of the hash function. Changing the width changes the result of the function and invalidates all existing entries in the hash table.

Adjusting the lock table to improve performance

The size of the hash table is set in the Firebird configuration file. You must shut down all activity on all databases that share the hash table – normally all databases on the machine – before changes take effect. The Classic architecture uses the lock table more heavily than SuperServer. If you choose the Classic architecture, you should check the load on the hash table periodically and increase the number of hash slots if the load is high. The symptom of an overloaded hash table is sluggish performance under load.

The tool for checking the lock table is fb_lock_print, which is a command line utility in the bin directory of the Firebird installation tree. The full lock print describes the entire state of the lock table and is of limited interest. When your system is under load and behaving badly, invoke the utility with no options or switches, directing the output to a file. Open the file with an editor. You'll see output that starts something like this:

```
LOCK_HEADER BLOCK
   Version:114, Active owner:0, Length:262144, Used:85740
   Semmask:0x0, Flags: 0x0001
   Enqs: 18512, Converts: 490, Rejects:0, Blocks: 0
   Deadlock scans:0, Deadlocks:0, Scan interval:10
   Acquires: 21048, Acquire blocks:0, Spin count:0
   Mutex wait:10.3%
   Hash slots:101, Hash lengths (min/avg/max):3/   15/   30
…
```

The seventh and eighth lines suggest that the hash table is too small and that it is affecting system performance. In the example, these values indicate a problem:

```
Mutex wait: 10.3%
Hash slots: 101, Hash lengths (min/avg/max):3/   15/   30
```

In the Classic architecture, each process makes its own changes to the lock table. Only one process is allowed to update the lock table at any instant. When updating the lock table, a process holds the table's mutex. A non-zero mutex wait indicates that processes are blocked by the mutex and forced to wait for access to the lock table. In turn, that indicates a performance problem inside the lock table, typically because looking up a lock is slow.

If the hash lengths are more than min 5, avg 10, or max 30, you need to increase the number of hash slots. The hash function used in Firebird is quick but not terribly efficient. It works best if the number of hash slots is prime.

Change this line in the configuration file:

```
#LockHashSlots = 101
```

Uncomment the line by removing the leading #. Choose a value that is a prime number less than 2048.

```
LockHashSlots = 499
```

The change will not take effect until all connections to all databases on the server machine shut down.

If you increase the number of hash slots, you should also increase the lock table size. The second line of the lock print

```
Version:114, Active owner 0,
Length: 262144, Used: 85740
```

tells you how close you are to running out of space in the lock table. The Version and Active owner are uninteresting. The length is the maximum size of the lock table. Used is the amount of space currently allocated for the various block types and hash table. If the amount used is anywhere near the total length, uncomment this parameter in the configuration file by removing the leading #, and increase the value.

```
#LockMemSize = 262144
```

to this

```
LockMemSize = 1048576
```

The value is bytes. The default lock table is about a quarter of a megabyte, which is insignificant on modern computers. Changing the lock table size will not take effect until all connections to all databases on the server machine shut down.

Inside BLOBs

Author: Dmitri Kouzmenko
kdv@ib-aid.com

Author: Alexey Kovyazin
ak@ib-aid.com

This is an excerpt from the book "1000 InterBase & Firebird Tips & Tricks" by Alexey Kovyazin and Dmitri Kouzmenko, which will be published in 2006.

How the server works with BLOBs

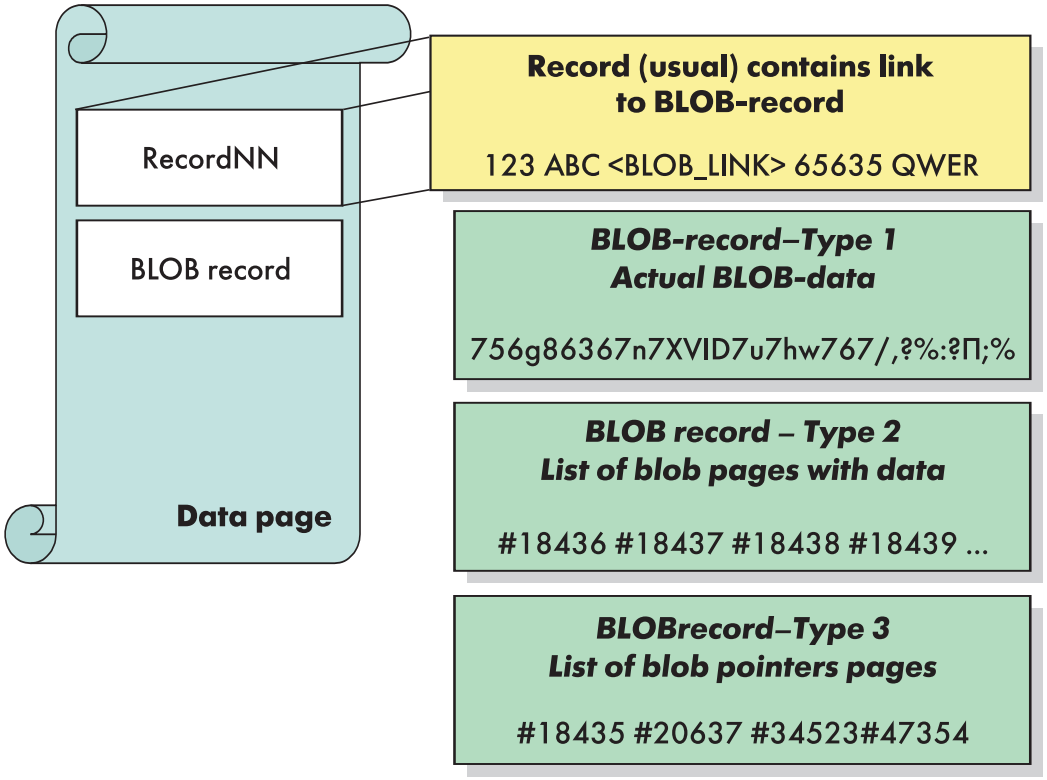
The BLOB data type is intended for storing data of variable size. Fields of BLOB type allow for storage of data that cannot be placed in fields of other types, - for example, pictures, audio files, video fragments, etc.

From the point view of the database application developer, using BLOB fields is as transparent as it is for other field types (see chapter "Data types" for details). However, there is a significant difference between the internal implementation mechanism for BLOBs and that for other data.

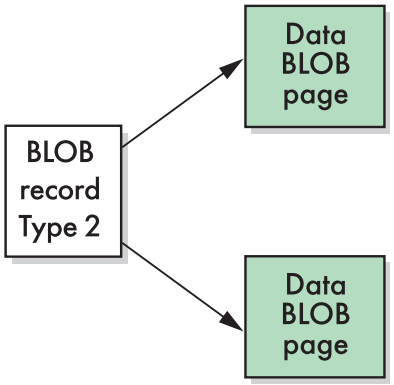
Unlike the mechanism used for handling other types of fields, the database engine uses a special mechanism to work with BLOB fields. This mechanism is transparently integrated with other record handling at the application level and at the same time has its own means of page organization. Let's consider in detail how the BLOB-handling mechanism works.

Initially, the basic record data on the data page includes a reference to a "BLOB record" for each non-null BLOB field, i.e. to record-like structure or quasi-record that actually contains the BLOB data. Depending on the size of the BLOB, this BLOB-record will be one of three types.

The first type is the simplest. If the size of BLOB-field data is less than the free space on the data page, it is placed on the data page as a separate record of "BLOB" type.

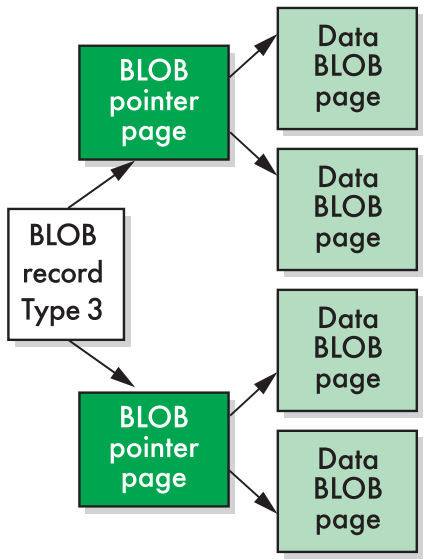


The second type is used when the size of BLOB is greater than the free space on the page. In this case, references to pages containing the actual BLOB data are stored in a quasi-record. Thus, a two-level structure of BLOB-field data is used.



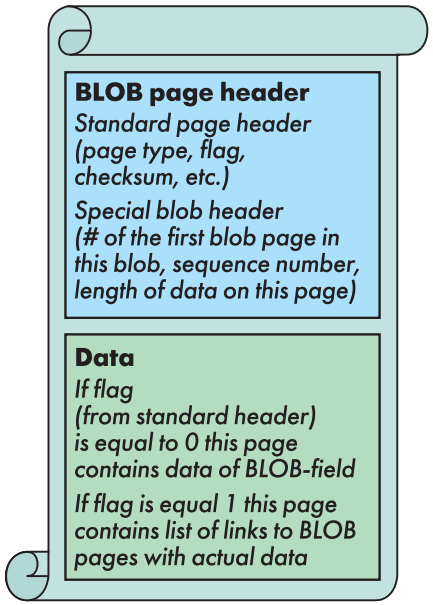
If the size of BLOB-field contents is very large, a three-level structure is used – a quasi-record stores references to BLOB pointer pages which contain references to the actual BLOB data.

The whole structure of BLOB storage (except for the quasi-record, of course) is implemented by one page type – the BLOB page type. Different types of BLOB-pages differ from each other in the presence of a flag (value 0 or 1) defining how the server should interpret the given page.



BLOB Page

The blob page consists of the following parts:



The special header contains the following information:

- The number of the first blob page in this blob. It is used to check that pages belong to one blob.
- A sequence number. This is important in checking the integrity of a BLOB. For a BLOB pointer page it is equal to zero.
- The length of data on a page. As a page may or may not be filled to the full extent, the length of actual data is indicated in the header.

Maximum BLOB size

As the internal structure for storing BLOB data can have only 3 levels of organization, and the size of data page is also limited, it is possible to calculate the maximum size of a BLOB.

However, this is a theoretical limit (if you want, you can calculate it), but in practice the limit will be much lower. The reason for this lower limit is that the length of BLOB-field data is determined by a variable of ULONG type, i.e. its maximal size will be equal to 4 gigabytes.

Moreover, in reality this practical limit is reduced if a UDF is to be used for BLOB processing. An internal UDF implementation assumes that the maximum BLOB size will be 2 gigabytes. So, if you plan to have very large BLOB fields in your database, you should experiment with storing data of a large size beforehand.

The segment size mystery

Developers of database applications often ask what the Segment Size parameter in the definition of a BLOB is, why we need it and whether or not we should set it when creating Blob-fields.

In reality, there is no need to set this parameter. Actually, it is a bit of a relic, used by the GPRE utility when pre-processing Embedded SQL. When working with BLOBs, GPRE declares a buffer of specified size, based on the segment size. Setting the segment size has no influence over the allocation and the size of segments when storing the BLOB on disk. It also has no influence on performance. Therefore the segment size can be safely set to any value, but it is set to 80 bytes by default.

Information for those who want to know everything: the number 80 was chosen because 80 symbols could be allocated in alphanumeric terminals.

Testing the NO SAVEPOINT feature in InterBase 7.5.1

Author: Vlad Horsun
hvlad@users.sourceforge.net

Author: Alexey Kovyazin
ak@ib-aid.com

In issue 1 we published Dmitri Yemanov's article about the internals of savepoints. While that article was still on the desk, Borland announced the release of InterBase 7.5.1, introducing, amongst other things, a NO SAVEPOINT option for transaction management. Is this an important improvement for InterBase? We decided to give this implementation a close look and test it some, to discover what it is all about.

Testing NO SAVEPOINT

In order to analyze the problem that the new transaction option was intended to address, and to assess its real value, we performed several very simple SQL tests. The tests are all 100% reproducible, so you will be able to verify our results easily.

Database for testing

The test database file was created in InterBase 7.5.1, page size = 4096, character encoding is NONE. It contains two tables, one stored procedure and three generators.

For the test we will use only one table, with the following structure:

```
CREATE TABLE TEST (
    ID            NUMERIC(18,2),
    NAME          VARCHAR(120),
    DESCRIPTION   VARCHAR(250),
    CNT           INTEGER,
    QRT           DOUBLE PRECISION,
    TS_CHANGE     TIMESTAMP,
    TS_CREATE     TIMESTAMP,
    NOTES         BLOB
);
```

This table contains 100,000 records, which will be updated during the test. The stored procedure and generators are used to fill the table with test data. You can increase the quantity of records in the test table by calling the stored procedure to insert them:

```
SELECT * FROM INSERTRECS(1000);
```

The second table, TEST2DROP, has the same structure as the first and is filled with the same records as TEST.

```
INSERT INTO TEST2DROP SELECT FROM TEST;
```

As you will see, the second table will be dropped immediately after connect. We are just using it as a way to increase database size cheaply: the pages occupied by the TEST2DROP table will be released for reuse after we drop the table. With this trick we avoid the impact of database file growth on the test results.

Setting test environment

All that is needed to perform this test is the trial installation package of InterBase 7.5.1, the test database and an SQL script.

Download the InterBase 7.5.1 trial version from www.borland.com. The installation process is obvious and well-described in the InterBase documentation.

You can download a backup of the test database ready to use from http://www.ibdeveloper.com/issue2/testspbackup.zip (~4 Mb) or, alternatively, an SQL script for creating it from http://www.ibdeveloper.com/issue2/testspdatabasescript.zip (~1 Kb).

If you download the database backup, the test tables are already populated with records and you can proceed straight to the section "Preparing to test", below.

If you choose instead to use the SQL script, you will create database yourself. Make sure you insert 100,000 records into table TEST using the INSERTRECS stored procedure and then copy all of them to TEST2DROP three or four times.

After that, perform a backup of this database and you will be on the same position as if you had down-

loaded "ready for use" backup.

Hardware is not a material issue for these tests, since we are only comparing performance with and without the NO SAVEPOINT option. Our test platform was a modest computer with Pentium-4, 2 GHz, with 512 RAM and an 80GB Samsung HDD.

Preparing to test

A separate copy of the test database is used for each test case, in order to eliminate any interference between statements. We create four fresh copies of the database for this purpose. Supposing all files are in a directory called C:\TEST, simply create the four test databases from your test backup file:

```
gbak –c –user SYSDBA –pass masterkey C:\TEST\testspbackup.gbk C:\TEST\testsp1.ib
gbak –c –user SYSDBA –pass masterkey C:\TEST\testspbackup.gbk C:\TEST\testsp2.ib
gbak –c –user SYSDBA –pass masterkey C:\TEST\testspbackup.gbk C:\TEST\testsp3.ib
gbak –c –user SYSDBA –pass masterkey C:\TEST\testspbackup.gbk C:\TEST\testsp4.ib
```

SQL test scripts

The first script tests a regular, one-pass update without the NO SAVEPOINT option.

For convenience, the important commands are clarified with comments:

```
connect "C:\TEST\testsp1.ib" USER "SYSDBA" Password "masterkey";  //Connect
drop table TEST2DROP;        // Drop table TEST2DROP to free database pages
commit;
select count(*) from test;        // Walk down all records in TEST to place them into cache
commit;
set time on;               //enable time statistics for performed statements
set stat on;              // enables writes/fetches/memory statistics
commit;
// perform bulk update of all records in TEST table
update TEST set ID = ID+1, QRT = QRT+1, NAME=NAME||'1', ts_change = CURRENT_TIMESTAMP;
commit;

quit;
```

The second script tests performance for the same UPDATE with the NO SAVEPOINT option:

News & Events

Fyracle 0.8.9

Janus has released a new version of Oracle-mode Firebird, Fyracle. Fyracle is a specialized build of Firebird 1.5: it adds temporary tables, hierarchical queries and a PL/SQL engine.

Version 0.8.9 adds support for stored procedures written in Java.

Fyracle dramatically reduces the cost of porting Oracle-based applications to Firebird.

Common usage includes the Compiere open source ERP package, mid-market deployments of Developer/2000 applications and demo CD's of applications without license trouble.

Read more at:

www.janus-software.com

```
connect "C:\testsp2.ib" USER "SYSDBA" Password "masterkey";
drop table TEST2DROP;
commit;
select count(*) from test;
commit;
set time on;
set stat on;
commit;
SET TRANSACTION NO SAVEPOINT;    // enable NO SAVEPOINT
update TEST set ID = ID+1, QRT = QRT+1, NAME=NAME||'1', ts_change = CURRENT_TIMESTAMP;
commit;
quit;
```

Except for the inclusion of the SET TRANSACTION NO SAVEPOINT statement in the second script, both scripts are the same, simply testing the behavior of engine in case of the single bulk UPDATE.

To test sequential UPDATEs, we added several UPDATE statements--we recommend using five.
The script for testing without NO SAVEPOINT would be:

```
connect "E:\testsp3.ib" USER "SYSDBA" Password "masterkey";
drop table TEST2DROP;
commit;
select count(*) from test;
commit;
set time on;
set stat on;
commit;
update TEST set ID = ID+1, QRT = QRT+1, NAME=NAME||'1', ts_change = CURRENT_TIMESTAMP;
update TEST set ID = ID+1, QRT = QRT+1, NAME=NAME||'1', ts_change = CURRENT_TIMESTAMP;
update TEST set ID = ID+1, QRT = QRT+1, NAME=NAME||'1', ts_change = CURRENT_TIMESTAMP;
update TEST set ID = ID+1, QRT = QRT+1, NAME=NAME||'1', ts_change = CURRENT_TIMESTAMP;
update TEST set ID = ID+1, QRT = QRT+1, NAME=NAME||'1', ts_change = CURRENT_TIMESTAMP;
commit;
quit;
```

You can download all the scripts and the raw results of their execution from this location:
http://www.ibdeveloper.com/issue2/testresults.zip

News & Events

IBAnalyst 1.9

IBSurgeon has issued new version of IBAnalyst.

Now it can better analyze InterBase or Firebird database statistics with using metadata information (in this case connection to database is required).

IBAnalyst is a tool that assists a user to analyze in detail Firebird or InterBase database statistics and identify possible problems with database performance, maintenance and how an application interacts with the database.

It graphically displays database statistics and can then automatically make intelligent suggestions about improving database performance and database maintenance

www.ibsurgeon.com/news.html

How to perform the test

The easiest way to perform the test is to use isql's INPUT command.

Suppose you have the scripts located in c:\test\scripts:

```
>isql
Use CONNECT or CREATE DATABASE to specify a database
SQL>input c:\test\scripts\script01.sql;
```

Test results

The single bulk UPDATE

First, let's perform the test where the single-pass bulk UPDATE is performed.

This is an excerpt from the one-pass script with default transaction settings.

```
SQL> update TEST set ID = ID+1, QRT = QRT+1, NAME=NAME||'1', ts_change = CURRENT
_TIMESTAMP;
Current memory = 10509940
Delta memory = 214016
Max memory = 10509940
Elapsed time= 4.63 sec
Buffers = 2048
Reads = 2111
Writes 375
Fetches = 974980

SQL> commit;
Current memory = 10340752
Delta memory = -169188
Max memory = 10509940
Elapsed time= 0.03 sec
Buffers = 2048
Reads = 1
Writes 942
Fetches = 3
```

This is an excerpt from the one-pass script with NO SAVEPOINT enabled

News & Events

CopyTiger 1.00

CopyTiger, a new product from Microtec, is an advanced database replication & synchronisation tool for InterBase/ Firebird, powered by their CopyCat component set (see the article about CopyCat in this issue).

For more information about CopyTiger see:

www.microtec.fr/copycat/ct

Download an evaluation version

```
SQL> SET TRANSACTION NO SAVEPOINT;
SQL> update TEST set ID = ID+1, QRT = QRT+1, NAME=NAME||'1', ts_change = CURRENT
_TIMESTAMP;
Current memory = 10352244
Delta memory = 55296
Max memory = 10352244
Elapsed time= 4.72 sec
Buffers = 2048
Reads = 2102
Writes 350
Fetches = 967154

SQL> commit;
Current memory = 10344052
Delta memory = -8192
Max memory = 10352244
Elapsed time= 0.15 sec
Buffers = 2048
Reads = 1
Writes 938
Fetches = 3
```

Performance appears to be almost the same, whether the NO SAVEPOINT option is enabled or not.

Sequential bulk UPDATE statements

With the mutlti-pass script (sequential UPDATEs) the raw test results are rather large.

IB 7.5.1 update 100k with default setting 5 times					
	Max mem, Kb	Time, ms	Writes	Reads	Delta mem, bytes
1	10271.926	4670	375	2111	214016
2	60427.926	478560	5941	2239	50305100
3	60427.926	159950	11831	2299	0
4	60427.926	156190	14688	2311	0
5	60427.926	160660	19317	2341	0

Table 1

Test results for 5 sequental UPDATEs

News & Events

SQLHammer 1.5 Enterprise Edition

The interesting developer tool, **SQLHammer**, now has an **Enterprise Edition**.

SQLHammer is a high-performance tool for rapid database development and administration.

It includes a common integrated development environment, registered custom components based on the Borland Package Library mechanism and an Open Tools API written in Delphi/Object Pascal.

www.metadataforge.com

IB 7.5.1 update 100k 5 times with NO SAVEPOINT					
	Max mem, Kb	Time, ms	Writes	Reads	Delta mem, bytes
1	10130.621	4950	354	2099	55296
2	10134.621	7210	5513	2220	0
3	10134.621	6230	11240	2298	0
4	10134.621	6380	14689	2312	0
5	10134.621	6560	18192	2341	0

Table 1

Test results for 5 sequental UPDATEs

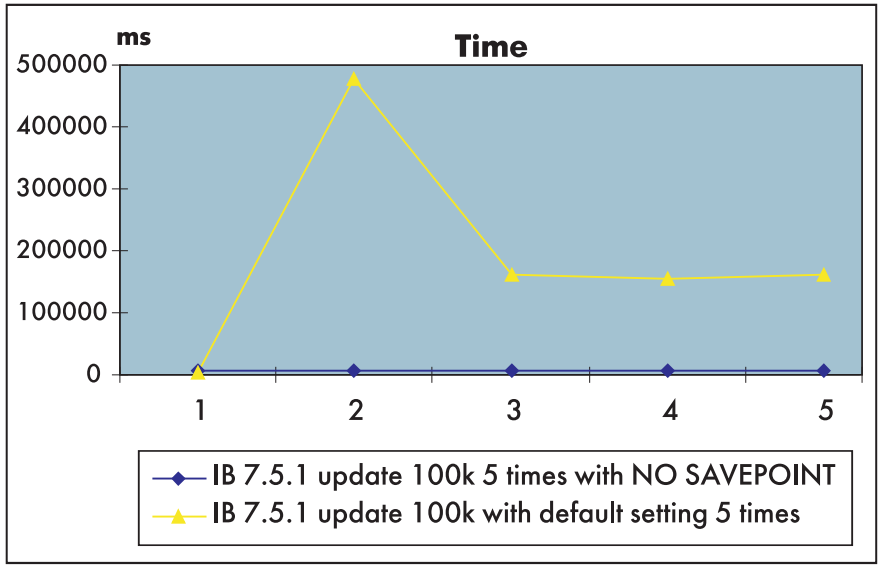


Figure 1

Time to perform UPDATEs with and without NO SAVEPOINT

For convenience, the results are tabulated above.

The first UPDATE statement has almost the same execution time with and without the NO SAVEPOINT option. However, memory consumption is reduced fivefold when we use NO SAVEPOINT.

In the second UPDATE we start to see the difference. With default transaction settings this UPDATE takes a very long time - 47 seconds - compared to only 7210 ms with NO SAVEPOINT enabled. With default transaction settings we can see that memory usage is significant, wherease with NO SAVEPOINT no additional memory is used.

The third and all following UPDATE statements with default settings show equal time and memory usage values and the growth of writes parameters.

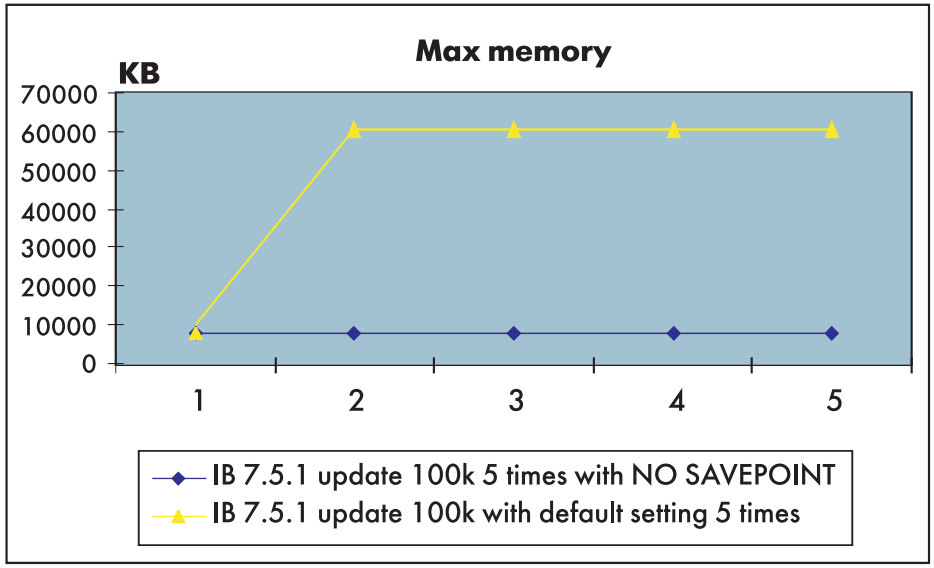


Figure 2

Memory usage while performing UPDATEs with and without NO SAVEPOINT

With NO SAVEPOINT usage we observe that time/memory values and writes growth are all small and virtually equal for each pass. The corresponding graphs are below:

Inside the UNDO log

So what happened during the execution of the test scripts? What is the secret behind this magic that NO SAVEPOINT does? Is there any cost attached to these gains?

A few words about versions

You probably know already that InterBase is a multi-record-version database engine, meaning that each time a record is changed, a new version of that record is produced. The

old version does not disappear immediately but is retained as a backversion.

In fact, the first time a backversion is written to disk, it is as a delta version, which saves disk space and memory usage by writing out only the differences between the old and the new versions. The engine can rebuild the full old version from the new version and chains of delta versions. It is only if the same record is updated more than once within the same transaction that the full backversion is written to disk.

The UNDO log concept

You may recall from Dmitri's article how each transaction is implicitly enclosed in a "frame" of savepoints, each having its own undo log. This log stores a record of each change in sequence, ready for the possibility that a rollback will be requested.

A backversion materializes whenever an UPDATE or DELETE statement is performed. The engine has to maintain all these backversions in the undo log for the relevant savepoint.

So, the Undo Log is a mechanism to manage backversions for savepoints in order to enable the associated changes to be rolled back. The process of Undo logging is quite complex and maintaining it can consume a lot of resources.

The NO SAVEPOINT option

The NO SAVEPOINT option in InterBase 7.5.1 is a workaround for the problem of performance loss during bulk updates that do multiple passes of a table. The theory is: if using the implicit savepoint management causes problems then let's kill the savepoints. No savepoints – no problem :-)

Besides ISQL, it has been surfaced as a transaction parameter in both DSQL and ESQL. At the API level, a new transaction parameter block (TPB) option isc_tpb_no_savepoint can be passed in the isc_start_transaction() function call to disable savepoints management. Syntax details for the latter flavors and for the new tpb option can be found in the 7.5.1 release notes.

The effect of specifying the NO SAVEPOINT transaction parameter is that no undo log will be created. However, along with the performance gain for sequential bulk updates, it brings some costs for transaction management.

First and most obvious is that, with NO SAVEPOINT enabled, any error handling that relies on savepoints is unavailable. Any error during a NO SAVEPOINT transaction precludes all subsequent execution and leads to rollback (see "Release Notes" for InterBase 7.5. SP1, "New in InterBase 7.5.1", page 2-2).

Secondly, when a NO SAVEPOINT transaction is rolled back, it is marked as rolled back in the transaction inventory page. Record version garbage thereby gets stuck in the "interesting" category and prevents the OIT from advancing. Sweep is needed to advance the OIT and back out dead record versions.

Fuller details of the NO SAVEPOINT option are provided in the InterBase 7.5.1. Release Notes.

Initial situation

Consider the implementation details of the undo-log. Figure 3 shows the initial situation:

Recall that we perform this test on freshly-restored database, so it is guaranteed that only one version exists for any record.

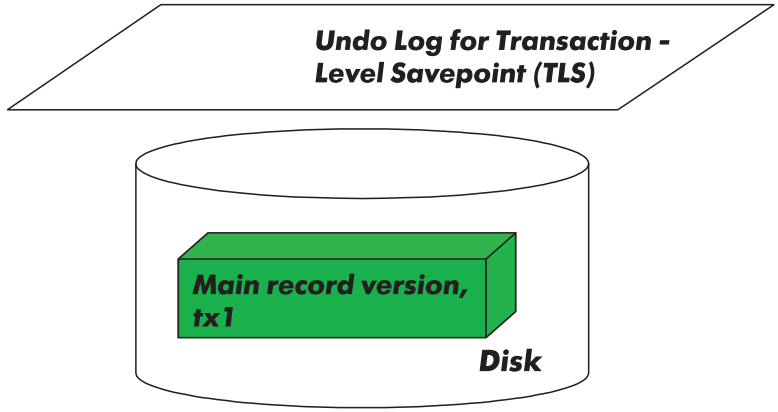


Figure 3

Initial situation before any UPDATE - only the one record version exists, Undo log is empty

The first UPDATE

The first UPDATE statement creates delta backversions on disk (see figure 4). Since deltas store only the differences between the old and new versions, they are quite small. This operation is fast and it is easy work for the memory manager.

It is simple to visualize the undo log when we perform the first UPDATE/DELETE statement inside the transaction – the engine just records the numbers of all affected records into the bitmap structure. If it needs to roll back the changes associated with this savepoint, it can read the stored numbers of the affected records, then walk down to the version of each and restore it from the updated version and the backversion stored on disk.

This approach is very fast and economical on memory usage. The engine does not waste too many resources to handle this undo log – in fact it reuses the existing multi-versioning mechanism. Resource consumption is merely the memory used to store the bitmap structure with the backversion numbers. We don't see any significant difference here between a transaction with the default settings and one with the NO SAVEPOINT option enabled.

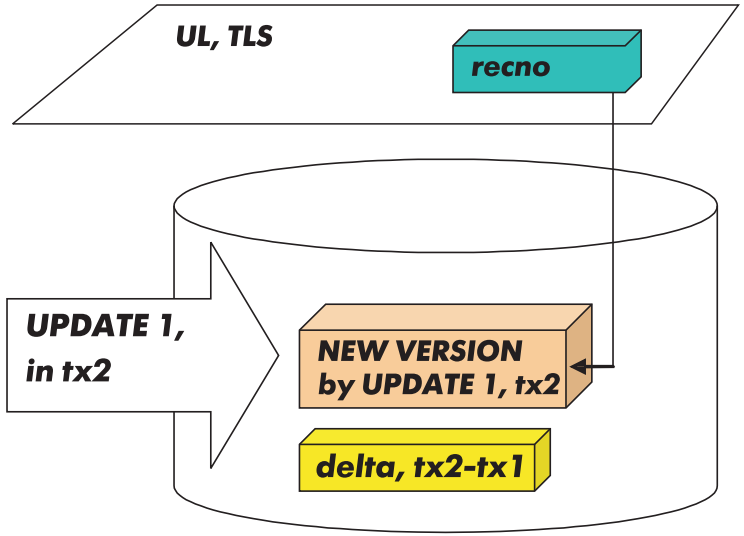


Figure 4

UPDATE1 create small delta version on disk and put record number into UNDO log

The second UPDATE

When the second UPDATE statement is performed on the same set of records, we have a different situation.

Here is a good place to note that the example we are considering is the most simple situation, where only the one global (transaction level) savepoint exists. We will also look at the difference in the Undo log when an explicit (or enclosed BEGIN... END) savepoint is used.

To preserve on-disk integrity (remember the 'careful write' principle ?) the engine must compute a new delta between the old version (by transaction1) and new version (by transaction2, update2), store it somewhere on disk, fetch the current full version (by transaction2, update1), put it into the in-memory undo-log, replace it with the new full version (with backpointers set to the newly created delta) and erase the old, now superseded delta. As you can see – there is much more work to do, both on disk and in memory.

The engine could write all intermediate versions to disk but there is no reason to do so. These versions are visible only to the modifying transaction and would not be used unless a rollback was required.

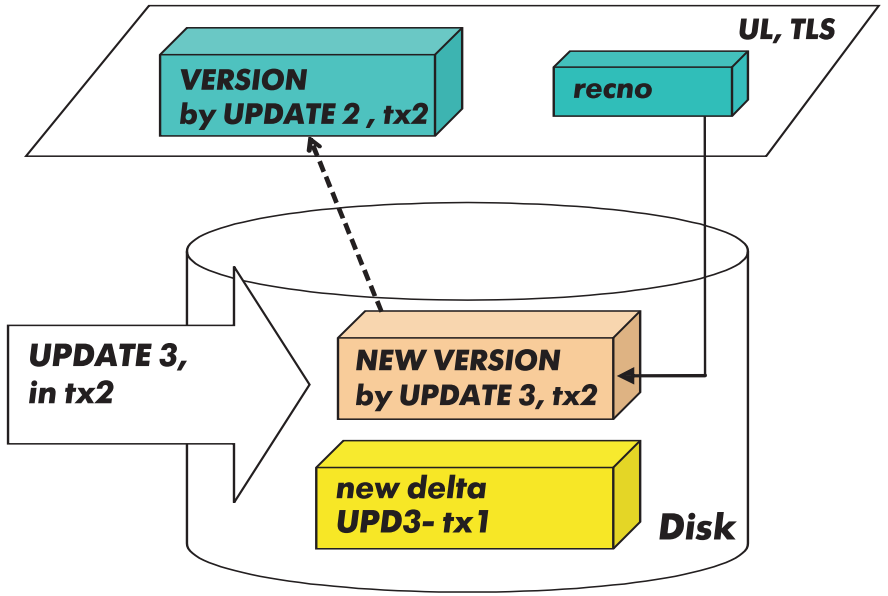


Figure 5

The second UPDATE creates a new delta backversion for transaction 1, erases from disk the delta version created by the first UPDATE, and copies the version from UPDATE1 into the Undo log.

This all makes hard work for the memory manager and the CPU, as you can see from the growth of the "max mem"\"delta mem" parameters values in the test that uses the default transaction parameters.

When NO SAVEPOINT is enabled we avoid the expense of maintaining the Undo log. As a result, we see execution time, reads/writes and memory usage as low for subsequent updates as for the first.

The third UPDATE

The third and all subsequent UPDATEs are similar to the second UPDATE, with one exception – memory usage does not grow any further.

Original design of IB implement second UPDATE another way but sometime after IB6 Borland changed original behavior and we see what we see now.But this theme is for another article;)

Why is the delta of memory usage zero? The reason is that, beyond the second UPDATE, no record version is created. From here on, the update just replaces record data on disk with the newest one and shifts the superseded version into the Undo log.

A more interesting question is why we see an increase in disk reads and writes during the test. We would have expected that the third and following UPDATEs would do essentially equal numbers of read and writes to write the newest versions and move the previous ones to the undo log. However, we are actually seeing a growing count of writes. We have no answer for it, but we would be pleased to know.

The following figure (figure 6) helps to illustrate the situation in the Undo log during the sequential updates. When NO SAVEPOINT is enabled, the only pieces we need to perform are replacing the version on disk and updating the original backversion. It is fast as the first UPDATE.

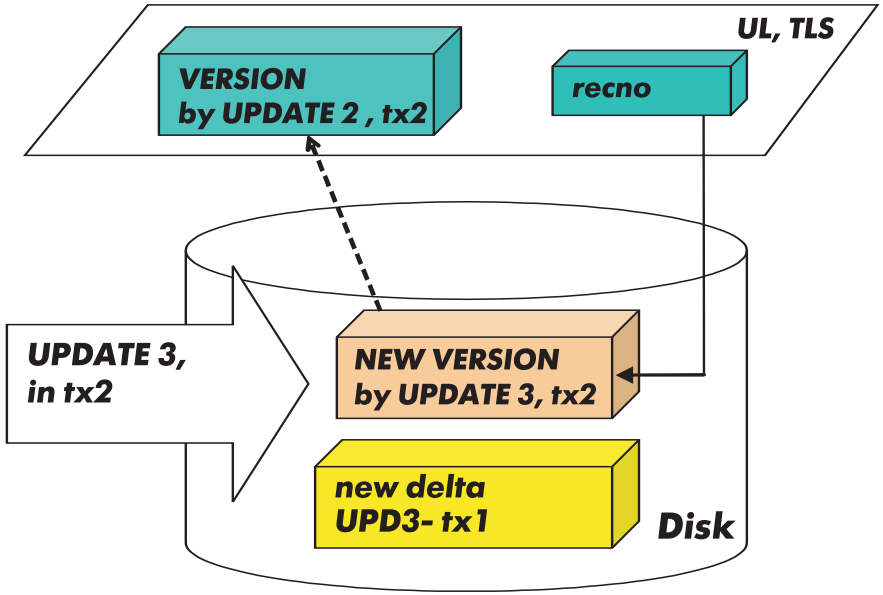


Figure 6

The third UPDATE overwrites the UPDATE1 version in the Undo log with the UPDATE2 version and its own version is written to disk as the latest one.

Explicit SAVEPOINT

When an UPDATE statement is going to be performed within its own explicit or implicit BEGIN... END savepoint framework, the engine has to store a backversion for each associated record version in the Undo log.

For example, if we used an explicit savepoint, e.g. SAVEPOINT Savepoint1, upon performing UPDATE2, we would have the situation illustrated in figure 7:

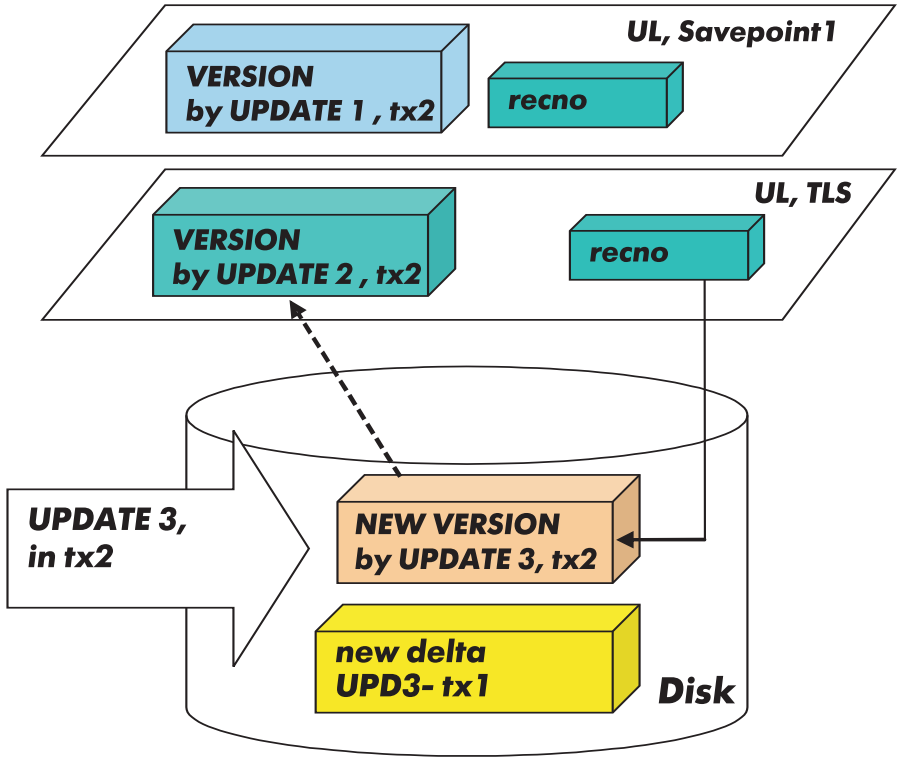


Figure 7

If we have an explicit SAVEPOINT, each new record version associated with it will have a corresponding backversion in the Undo log of that savepoint

In this case the memory consumption would be expected to increase each time an UPDATE occurs within the explicit savepoint's scope.

Summary

The new transaction option NO SAVEPOINT can solve the problem of excessive resource usage growth that can occur with sequential bulk updates. It should be beneficial when applied appropriately. Because the option can create its own problems by inhibiting the advance of the OIT, it should be used with caution, of course. The developer will need to take extra care about database housekeeping, particularly with respect to timely sweeping.

Object-Oriented Development in RDBMS, Part 1

Author: Vladimir Kotlyarevsky
vlad@contek.ru

Thanks and apologies

This article is mostly a compilation of methods that are already well-known, though many times it turned out that I on my own have reinvented a well-known and quite good wheel. I have endeavored to provide readers with links to publications I know of that are discussing the problem. However, if I missed someone's work in the bibliography, and thus violated copyright, please drop me a message at vlad@contek.ru. I apologize beforehand for possible inconvenience, and promise to add any necessary information to the article.

The sources in the bibliography are listed in the order of their appearance in my mind.

The described database structures have been simplified in order to illustrate the problems under consideration as much clearly as possible, leaving out more unimportant elements. I have tested the viability of all the methods taken from the articles, and of course I have tested all my own methods.

Mixing of object-oriented programming and RDBMS use is always a compromise. I have endeavored to recommend several approaches in which this compromise is minimised for both components. I have also tried to describe both advantages and disadvantages of such a compromise.

I should make it clear that the object approach to database design as described is not appropriate for every task. It is still true for the OOP as a whole, too, no matter what OOP apologists may say! ?. I would recommend using it for such tasks as document storage and processing, accounting, etc.

And the last, but not least , I am very thankful to Dmitry Kuzmenko, Alexander Nevsky and other people who helped me in writing this article.

The problem statement

What is the problem?

Present-day relational databases were developed in times when the sun shone brighter, the computers were slower, mathematics was in favour, and OOP had yet to see the light of day. Due to that fact most RDBMSs' have the following characteristics in common:

1.Everyone got used to them and felt comfortable with them.

2.They are quite fast (if you use them according to certain known standards).

3.They use SQL, which is an easy, comprehensible and time-proved data manipulation method.

4.They are based upon a strong mathematical theory.

5.They are convenient for application development - if you develop your applications just like 20-30 years ago.

As you see, almost all these characteristics sound good, except, probably, the last one. Today you can hardly find a software product (in almost any area) consisting of more than few thousand of lines which is written without OOP technologies. OOP languages have been used for a long time for building visual forms, i.e. in UI development. It is also quite usual to apply OOP at the business logic level, if you implement it either on a middle-tier, or on a client. But things fall apart when the deal comes closer to the data storage issues... During the last ten years there were several attempts to develop an object-oriented database system, and, as far as I know, all those attempts were rather far from being successful. The characteristics of an OODBMS are the antithesis of those for an RDBMS. They are unusual and

slow; there are no standards for data access and no underlying mathematical theory. Perhaps the OOP developer feels more comfortable with them, although I am not sure...

As a result, everyone continues using RDBMS, combining object-oriented business logic and domain objects with relational access to the database, where these objects are stored.

What do we need?

The thing we need is simple – to develop a set of standard methods that will help us to simplify the process of tailoring the OO-layer of business logic and a relational storage together. In other words, our task is to find out how to store objects in a relational database, and how to implement links between the objects. At the same time we want to keep all the advantages provided by the relational database design and access: speed, flexibility, and the power of relation processing.

RDBMS as an object storage

First let's develop a database structure that would be suitable for accomplishing the specified task.

The OID

All objects are unique, and they must be easily identifiable. That is why all the objects stored in the database should have unique ID-keys from a single set (similar to object pointers in run-time). These identifiers are used to link to an object, to load an object into a run-time environment, etc. In the [1] article these identifiers are called OIDs (i.e. Object IDs), in [2] – UINs (Unique Identification Number), or "hyperkey". Let us call them OIDs, though "hyperkey" is also quite a beautiful word, isn't it? ? .

First of all, I would like to make a couple of points concerning key uniqueness. Database developers who are used to the classical approach to database design would probably be quite surprised at the idea that sometimes it makes sense to make a table key unique not only within a single table (in terms of OOP – not only within a certain class), but also within the whole database (all classes). However, such strict uniqueness offers important advantages, which will become obvious quite soon. Moreover, it often makes sense to provide complete uniqueness in a Universe, which provides considerable benefits in distributed databases and replication development. At the same time, strict uniqueness of a key within the database does not have any disadvantages. Even in the pure relational model it does not matter whether the surrogate key is unique within a single table or the whole database.

OIDs should never have any real world meaning. In other words, the key should be completely surrogate. I will not list here all the pros and cons of surrogate keys in comparison with natural ones: those who are interested can refer to the [4] article. The simplest explanation is that everything dealing with the real world may change (including the vehicle engine number, network card number, name, passport number, social security card number, and even sex ?.

Nobody can change their date of birth – at least not their de facto date of birth. But birth dates are not unique, anyway.)

Remember the maxim "everything that can go bad will go bad" ("consequently, everything that cannot go bad...". hum!. But let's not talk about such gloomy things ?). Changes to some OIDs would immediately lead to changes in all identifiers and links, and thus, as Mr. Scott Ambler wrote [1], could result in a "huge maintenance nightmare." As for the surrogate key, there is no need to change it, at least in terms of dependency on the changing world.

And what is more, nobody requires run-time pointers to contain some additional information about an object except for a memory address. However, there are some people who vigorously reject usage of surrogates. The most brilliant argument against surrogates I've ever heard is that "they conflict with the relational theory». This statement is quite arguable, since surrogate keys, in some sense, are much closer to that theory than natural ones.

Those who are interested in more strong evidence supporting the use of OIDs with the characteristics described above (pure surrogate, unique at least within the database), should refer to [1], [2], and [4] articles.

The simplest method of OID implementation in a relational database is a field of "integer" type, and a function for generating unique values of this type. In larger or distributed databases, it probably makes sense to use "int64" or a combination of several integers.

ClassId

All objects stored in a database should have a persistent analogue of RTTI, which must be immediately available through the object identifier. Then, if we know the OID of an object, keeping in mind that it is

unique within the database, we can immediately figure out what type the object is. This is the first advantage of OID uniqueness. Such an objective may be accomplished by a ClassId object attribute, which refers to the known types list, which basically is a table (let us call it "CLASSES"). This table may include any kind of information – from just a simple type name to detailed type metadata, necessary for the application domain.

Name, Description, creation_date, change_date, owner

Imagine a file system where user has to remember the handles or the inodes of files instead of their filenames ?. It is frequently convenient, though not always necessary, to have an object naming method that is independent of the object type. For example, each object may have a short name and a long name. It is also sometimes convenient to have other object attributes for different purposes, akin to file attributes, such as "creation_date," "change_date," and "owner" (the "owner" attribute can be a string containing the owner's name, as well as a link to an object of the "user" type). The "Deleted" attribute is also necessary as an indicator of the unavailability of an object. The physical deletion of records in a database, full of direct and indirect links, is often a very complicated task, to put it mildly ?.

Thus each object has to have mandatory attributes ("OID" and "ClassId") and desirable attributes ("Name," "Description," "creation_date," "change_date," and "owner"). Of course, when developing an application system, you can always add extra attributes specific to your particular requirements. This issue will be considered a little later.

The OBJECTS Table

Reading this far leads us to the conclusion that the simplest solution is to store the standard attributes of all objects in a single table. You could support a set of these fixed attributes separately for each type, duplicating 5 or 6 fields in each table, but why? It is not a duplication of information, but it is still a duplication of entities. Besides, a single table would allow us to have a central "well-known" entry point for searching of any object, and that is one of the most important advantages of the described structure.

So let us design this table:

```
create domain TOID as integer not null;
create table OBJECTS(
   OID TOID primary key,
   ClassId TOID,
   Name varchar(32),
   Description varchar(128),
   Deleted smallint,
   Creation_date timestamp default CURRENT_TIMESTAMP,
   Change_date timestamp default CURRENT_TIMESTAMP ,
   Owner TOID);
```

The ClassId attribute (the object type identifier) refers to the OBJECTS table, that is, the type description is also an object of a certain type, which has, for example, well-known ClassId = –100. (You need to add a description of the known OID). Then the list of all persistent object types that are known to the system is sampled by a query: select OID, Name, Description from OBJECTS where ClassId = -100).

Each object stored in our database will have a single record in the OBJECTS table referring to it, and hence a corresponding set of attributes. Wherever we see a link to a certain unidentified object, this fact about all objects enables us to find out easily what that object is - using a standard method.

Quite often, only basic information is needed. For instance, we need just a name to display in a lookup combobox. In this case we do not need anything but the OBJECTS table and a standard method of obtaining the name of an object via the link to it. This is the second advantage.

There are also types, simple lookup dictionaries, for example, which do not have any attributes other than those which already are in OBJECTS. Usually it is a short name (code) and a full long name that can easily be stored in the "Name" and "Description" fields of the OBJECTS table. Do you remember how many simple dictionaries are in your accounting system? It is likely, that not less than a half! Thus you may consider that you have implemented half of these dictionaries - that is the third advantage! Why should different dictionaries (entities) be stored in different tables (relations), when they have the same attributes? It does not matter that you were told to do so when you were a student!

A simple plain dictionary can be retrieved from the OBJECTS table by the following query

```
select OID, Name, Description
   from OBJECTS
   where ClassId = :LookupId
```

if you know ClassId for this dictionary element. If you only know the type name, the query becomes a bit more complex:

```
select OID, Name, Description
    from OBJECTS
    where ClassId = (select OID from OBJECTS where ClassId = -100
    and Name = :LookupName)
```

Later I will demonstrate how to add a little more intelligence to such simple dictionaries.

Storing of more complex objects

It is clear that some objects in real databases are more complex than those which can be stored in the OBJECTS table. The method for storing them depends on the application domain and the object's internal structure. Let's look at three well-known methods of object-relational mapping.

Method 1. The objects are stored just as in a standard relational database, with the type attributes mapped to table attributes. For example, document objects of the "Order" type with such attributes as "order number," "comments," "customer," and "order amount" are stored in the table Orders

```
create table Orders (
    OID TOID primary key,
    customer TOID,
    sum_total NUMERIC(15,2)),
```

which relates one-to-one to the OBJECTS table by the OID field. The "order number" and "comments" attributes are stored in the "Name" and "Description" fields of the OBJECTS table. "Orders" also refers to "OBJECTS" via the "customer" field, since a customer is also an object, being for example, a part of the "Partners" dictionary. You can retrieve all attributes of "Order" type with the following query:

```
select o.OID,
 o.Name as Number,
 o.Description as Comment,
 ord.customer,
 ord.sum_total
 from Objects o, Orders ord
 where o.OID = ord.OID and ord.OID = :id
```

As you see, everything is simple and usual. You could also create a view "orders_view", and make everything look as it always did. ?

If an order has a "lines" section, and a real-world order should definitely have such a section, we can create separate table for it, call it e.g. "order_lines", and relate it with the "Orders" table by a relation 1:M.

```
create table order_lines (
 id integer not null primary key,
 object_id TOID, /* reference to order object - 1:M relation */
 item_id TOID, /* reference to ordered item */
 amount numeric(15,4),
 cost numeric(15,4),
 sum numeric(15,4) computed by (amount*cost))
```

One very important advantage of this storage method is that it allows you to work with object sets as you would with normal relational tables (which they actually are). All the advantages of the relational approach are present.

Nevertheless, there are two main disadvantages: implementation of the system of object-relational mapping for this method is less than simple, and there are some difficulties in the organization of type inheritance. This method is described in detail in [1] and [3]. These articles also describe the implementation of type inheritance methods in a database.

Method 2. (See. [5]) All object attributes of any type are stored in the form of a record set in a single table. A simple example would look like this:

```
create table attributes (
 OID TOID, /* link to the master-object of this attribute */
 attribute_id integer not null,
 value varchar(256),
 constraint attributes_pk primary key (OID, attribute_id));
```

connected 1:M with OBJECTS by OID. There is also a table

```
create table class_attributes (
 OID TOID, /*here is a link to a description-object of the type    */
 attribute_id integer not null,
 attribute_name varchar(32),
 attribute_type integer,
 constraint class_attributes_pk primary key (OID,attribute_id))
```

which describes type metadata – an attribute set (their names and types) for each object type.

All attributes for particular object where the OID is known are retrieved by the query:

```
select attribute_id, value
 from attributes
 where OID = :oid
```

or, with names of attributes

```
select a.attribute_id, ca.attribute_name a.value
 from attributes a, class_attributes ca, objects o
 where a.OID = :oid and
 a.OID = o.OID and
 o.ClassId = ca.OID and
 a.attribute_id = ca.attribute_id
```

In the context of this method, you can also emulate a relational standard. Instead of selecting object attributes in several records (one attribute per a record) you can get all attributes in a single record by joining or by using subqueries:

```
select o.OID,
    o.Name as Number,
    o.Description as Comment,
    a1.value as customer,
    a2.value as sum_total
    from OBJECTS o
    left join attributes a1 on a1.OID = o.OID and
    a1.attribute_id = 1
    left join attributes a2 on a2.OID = o.OID and
    a2.attribute_id = 2
    where o.OID = :id;
```

Clearly, the more attributes object has, the slower the loading process will be, since each attribute requires an additional join in the query.

Returning to the example with the "Order" document, we see that the "order number" and "comments" attributes are still stored in the OBJECTS table, but "customer" and "order amount" are stored in two separate records of the "attributes" table. This approach is described by Anatoliy Tentser in article [5]. Its advantages are rather important: a standardized method of retrieving and storing object attributes; ease of extending and changing a type; ease in implementing object inheritance; very simple structure for the database, a significant benefit benefit since, with this approach the number of tables would not increase, no matter how many different types were stored in a database.

The main disadvatage is that this method is so different from the standard relational model that many standard techniques used on relational databases cannot be applied. The speed of object retrieval is significantly lower than the previous method 1 and it decreases also with the growth of the attribute count of a type. It is not very suitable for working with objects using SQL from inside the database, in stored procedures, for example. There is also a certain level of data redundancy because three fields ("OID," "attribute_id," and "value") are applicable to every attribute instead of just the one field described in Method 1.

Method 3. Everything is stored in BLOB, and one of the persistent formats is applied – a custom format, or, for example, dfm (VCL streaming) from the Borland VCL, or XML, or anything you like. There is nothing to comment on here. The advantages are obvious: object retrieval logic is simple and fast; no extra database structures are necessary – just a single BLOB field; you can store any custom objects, including absolutely unstructured objects (such as MS Word documents, HTML pages, etc). The disadvantage is also obvious: there is nothing relational about this approach and you would have to perform all data processing outside of the database, using the database only for object storage.

Editor's note: *Even more important, it is impossible to query the database for objects with certain attributes, as the attribute data is stored in the BLOB.*

It was not difficult to come to the following conclusions (well I know – everyone already knows them ?). All the three methods described are undoubtedly useful and have a right to live. Moreover, it sometimes makes sense to use all three of them in a single application. But when choosing among these three methods, take into account the listed advantages and disadvantages of each method. If your application involves massive relational data processing like searching or grouping, or it is likely to be added in the future, using a certain object attributes, it would be better to to use method 1, since it is the closest to the standard relational model and you will retain all the power of SQL. If, on the other hand, data processing is not particularly complex and data sets are not too large and/or you need a simple database structure, then it makes sense to use method 2. If a database is used only as an object storage, and all operations are performed with run-time instances of objects without using native database tools like SQL (except work with attributes stored in the OBJECTS table), the third method would probably be the best choice due to the speed and ease of implementation.

To be continued...

Replicating and synchronizing InterBase/Firebird databases using CopyCat

Author:
Jonathan Neve
jonathan@microtec.fr

www.microtec.fr

PART 1 : BASICS OF DATABASE REPLICATION

A replicator is a tool for keeping several databases synchronized (either wholly or in part), on a continuous basis. Such a tool can have many applications: it can allow for off-line, local data editing, with a punctual synchronization upon reconnection to main database; it can also be used over a slow connection, as an alternative to a direct connection to the central database; another use would be to make an automatic, off-site, incremental backup, by using simple one-way replication.

Creating a replicator can be quite tricky. Let's examine some of the key design issues involved in database replication, and explain how these issues are implemented in Microtec CopyCat, a set of Delphi / C++Builder components for performing replication between Interbase and FireBird databases.

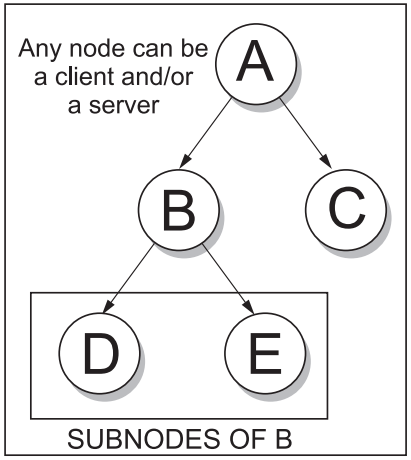
Data logging

Before anything can be replicated, all changes to each database must of course be logged. CopyCat creates a log table and triggers for each table that is to be replicated. These triggers insert into the log table all the information concerning the record that was changed (table name, primary key value(s), etc).

Multi-node replication

Replicating to and from several nodes adds another degree of complexity. Every change that is made to one database must be applied to all the others. Furthermore, when one database applies this change, it must indicate to the originating database that the change has been applied, without in any way hindering the other databases from replicating the same change, either before, simultaneously, or after.

In CopyCat, these problems are solved using a simple and flexible system. Each replication node can be have one parent node, and several sub-nodes towards which it



replicates its changes. Each node's list of sub-nodes is stored in a table in the node's database. (Incidentally, the parent node is configured in the replicator software itself rather than in the database, and therefore, no software is needed on nodes having no parent – which allows these servers to run Linux, or any other OS supported by Interbase/FireBird).

When a data change occurs in a replicated table, one line is generated per sub-node. Thus, each sub-node fetches only the log lines that concern it.

Two-way replication

One obvious difficulty involved in two-way replication is how to avoid changes that have been replicated to one database from replicating back to the original database. Since all the changes to the database are logged, the changes made by the replicator are also logged, and will therefore bounce back and forth between the source and the target databases. How can this problem be avoided?

The solution CopyCat uses is related to the sub-node management system described above. Each sub-node is assigned a name, which is used when the sub-node logs in to the database. When a sub-node replicates its own changes to its parent, the replication triggers log the change for all the node's sub-nodes except the current user. Thus, only sub-nodes other than the originator receive the change.

Conversely, CopyCat logs in to the nodes local database using the node name of its parent as user name. Thus, any change made to the local database during replication will be logged for all sub-nodes other than the node's parent, and any change made to the parent node will be logged to other sub-nodes, but not to the originating node itself.

Primary key synchronization

One problem with replication is that since data is edited off-line, there is no centralized way to ensure that the value of a field remains unique. One common answer to this problem is to use GUID values. This is a good solution if you're implementing a new database (except that GUID fields are rather large, and therefore, not very well suited for a primary key field), but if you have an exist-

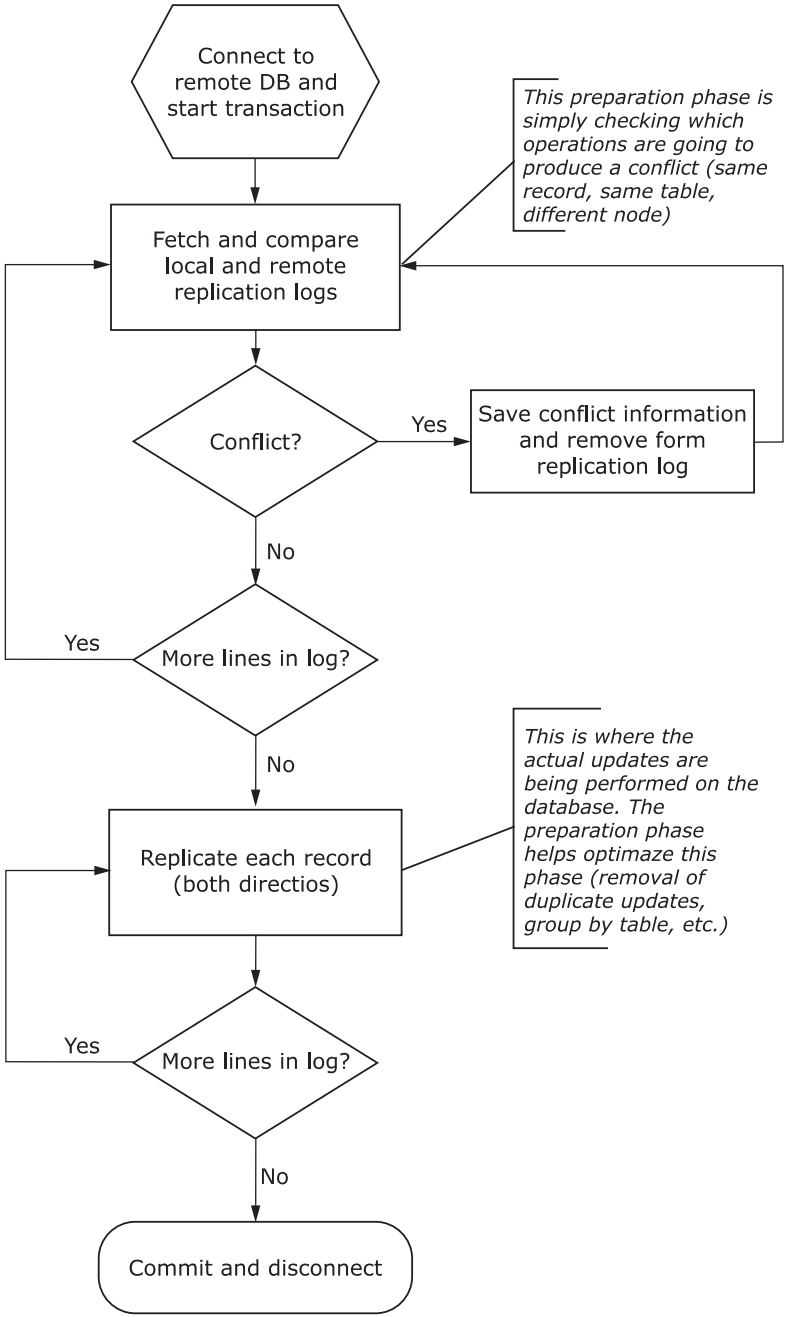
ing database that needs replication, it would be very difficult to replace all primary or unique key fields by GUID values.

Since GUID fields are, in many cases, not feasible, CopyCat implements another solution. CopyCat allows you to define for each primary key field (as well as up to three other fields for which unicity is to be maintained) a synchronization method. In most cases, this will be either a generator, or a stored procedure call, though it could be any valid SQL clause. Upon replication, this SQL statement is called on the server side in order to calculate a unique key value, and the resulting value is then applied to the local database. Only after the key values (if any) have been changed locally is the record replicated to the server.

When replicating from the parent node to the local node however, this behaviour does not take place: the primary key values on the server are considered to be unique.

Conflict management

Suppose a replication node and its parent both modify the same record during the same time period. When the replicator connects to its parent to replicate its changes, it has no way of telling which of the two nodes has the most up-to-date version of the record: this is a conflict.



CopyCat automatically detects conflicts, logs them to a dedicated table, and disables replication of that record in either direction until the conflict is resolved. The conflicts table holds the user names of both nodes involved in the conflict, as well as a field called "CHOSEN_USER". In order to solve the conflict, the user simply has to put in this field the name of the node which has the correct version of the record, and automatically, upon the next replication, the record will be replicated and the conflict resolved.

This system was carefully designed to function correctly even in some of the complex scenarios that are possible with CopyCat. For instance, the conflict may in reality be between two nodes that are not directly connected to each other: since CopyCat nodes only ever communicate directly with their parent, there is no way to tell if another node may not have a conflicting update for a certain record. Furthermore, it's entirely possible that two nodes (having the same parent) should simultaneously attempt to replicate the same record to their parent. By using a snapshot-type transaction, and careful ordering of the replication process, these issues are handled transparently.

Difficult database structures

There are certain database architectures that are difficult to replicate. Consider for example a "STOCK" table, containing one line per product, and a field holding the current stock value. Suppose that for a certain product, the current stock value being 45, node A adds 1 item to stock, setting the stock value to 46. Simultaneously, node B, adds 2 items to stock thereby setting the current stock value to 47. How can such a table then be replicated? Neither A nor B have the correct value for the field, since neither take into consideration the changes from the other node.

Most replicators would require such an architecture to be altered. Instead of having one record hold the current stock value of product, there could be one line per change. This would solve the problem. However, restructuring large databases (and the end-user applications that usually go with them) could be a rather major task. CopyCat was specifically designed to avoid these problems altogether, rather than require the database structure to be changed.

To solve this kind of problem, CopyCat introduces stored procedure "replication". That is, a mechanism for logging stored procedure calls, and replicating them to other nodes. When dealing with an

unreplicatable table (like in the example above) one solution is to make a stored procedure which can be called for updating the table, and using stored procedure replication in order to replicate each of these calls. Thus, continuing the example above, instead of replicating the values of the STOCK table, the nodes would replicate only their changes to these values, thereby correctly synchronizing and merging the changes to the STOCK table.

PART 2: GETTING STARTED WITH COPYCAT

Copycat is available in two distinct forms :

1. As a set of Delphi / C++ Builder components

2. As a standalone Win32 replication tool

We will now present each one of these products.

1. CopyCat Delphi / C++ Builder Component Set

CopyCat can be obtained as a set of Delphi / C++ Builder components, enabling you to use replication features painlessly and in many different situations, and sparing you the tedious task of writing and testing custom database synchronization solutions.

Using these components, replication or synchronization facilities can be seamlessly built into existing solutions. As an example, we have used the CopyCat components in an application used by our development team on their laptops to keep track of programming tasks. When the developers need to go on-site to visit a customer, the application runs in local mode on the laptop. When they return and connect up to the company network, any changes they have made on-site are automatically synchronized with the main Interbase database running on a Linux server.

Many more applications are possible since the CopyCat components are very flexible and allow for synchronization of even a single table!

Below is a concise guide for getting started with the CopyCat component suite:

Download and install the evaluation components from http://www.microtec.fr/copycat

Prepare databases

1) Open Delphi, and compile the data provider package(s) for the DAC that you plan to use (currently IBX and FIBPlus are supported). These are components for interfacing between the CopyCat components and the underlying data-access components.

2) Open and run the "Configuration" example project (requires the IBX provider).

3) On the "General" tab, fill in the connection parameters, and press "Connect to Database".

4) On the "Users" tab, provide the list of sub-nodes for the current database.

5) On the "Tables" tab, for each table that you want to replicate, set a priority (relative to the other tables), and double-click on the "PKn generator"

columns to (optionally) fill the primary key synchronization method. Once these settings have been made, set the "Created" field to 'Y', so as to generate the meta-data.

6) On the "Procedures" tab, set "Created" to 'Y' for each procedure that you want to replicate, after having set a priority.

7) Apply all the generated SQL to all databases that should be replicated.

8) For each database to be replicated, set the list of sub-nodes (in the RPL$USERS table).

Replicate

1. In Delphi, open the "Replicator" example project.

2. Drop a provider component on the form, and hook it up to the TCcReplicator's DBProvider property.

3. Setup the LocalDB and RemoteDB properties of the TCcReplicator with the connection parameters for the local and remote databases.

4. Fill in the user name of the local and remote nodes, as well as the SYSDBA user name and password (needed for primary key synchronization).

5. Compile and run the example.

6. Press the "Replicate now" button.

2. CopyTiger : the CopyCat Win32 standalone replication tool

For those who have replication or synchronization needs but do not have the time or resources to develop their own integrated system, Microtec proposes a standalone database replication tool based on the CopyCat technology called COPYTIGER.

Features include :

- Easy to use installer
- Independent server Administrator tool (CTAdmin)
- Configuration wizard for setting up links to master / slave databases
- Robust replication engine based on Microtec CopyCat
- Fault-tolerant connection mechanism allowing for automatic resumption of lost database connections
- Simple & intuitive control panel
- Automatic email notification on certain events (conflicts, PK violations, etc.)

Visit the CopyTiger homepage to download a time-limited trial version.

http://www.microtec.fr/copycat/ct

SUMMARY

In today's connected world, database replication and synchronization are topics of great interest among all industry professionals. With the advent of Microtec CopyCat, the Interbase / Firebird community is obtaining a two-fold benefit :

1. By encapsulating all the functionality of a replicator into Delphi components, CopyCat makes it easier than ever to integrate replication and synchronization facilities into custom applications,

2. By providing a standalone tool for the replication of Interbase / Firebird databases, Microtec is responding to another great need in the community – that of having a powerful and easy-to-use replication tool, and one that can be connected to an existing database without disrupting it's current structure.

CopyCat is being actively developed by Microtec and many new features are being worked on such as support for replicating between heterogeneous database types (PostgreSQL, Oracle, MSSQL, MySQL, NexusDB, ...) as well as a Linux / Kylix version for the components and the standalone tool.

You can find more information about CopyCat at:
http://www.microtec.fr/copycat, or by contacting us at copycat@microtec.fr.

Understanding Your Database with IBAnalyst

Author: Dmitri Kouzmenko
kdv@ib-aid.com

I have been working with InterBase since 1994. Back then, most databases were small, and did not require any tuning. Of course, there were occasions when I had to change ibconfig on a server, and reconfigured hardware or the OS, but that was almost all I could do to tune performance.

Four years ago, our company began to provide technical support and training of InterBase users. Working with many production databases also taught me a lot of different things. However, most of what I learned concerned applications – transaction parameters usage, optimizing queries and result sets.

Of course, I had known for quite some time about gstat – the tool that gives database statistics information. If you have ever looked in gstat output or read opguide.pdf about it, you would know that the statistical output looks like just a bunch of numbers and nothing else. Ok, you can discover fragmentation information for a particular table or index, but what other useful information can be obtained?

Thankfully, prior to working with InterBase, I was interested in different data structures, how they are stored and what algorithms they use. This helped me to interpret the output of gstat. At that time I decided to write a tool that could analyze gstat output to help in tuning the database or at least to identify the cause of performance problems.

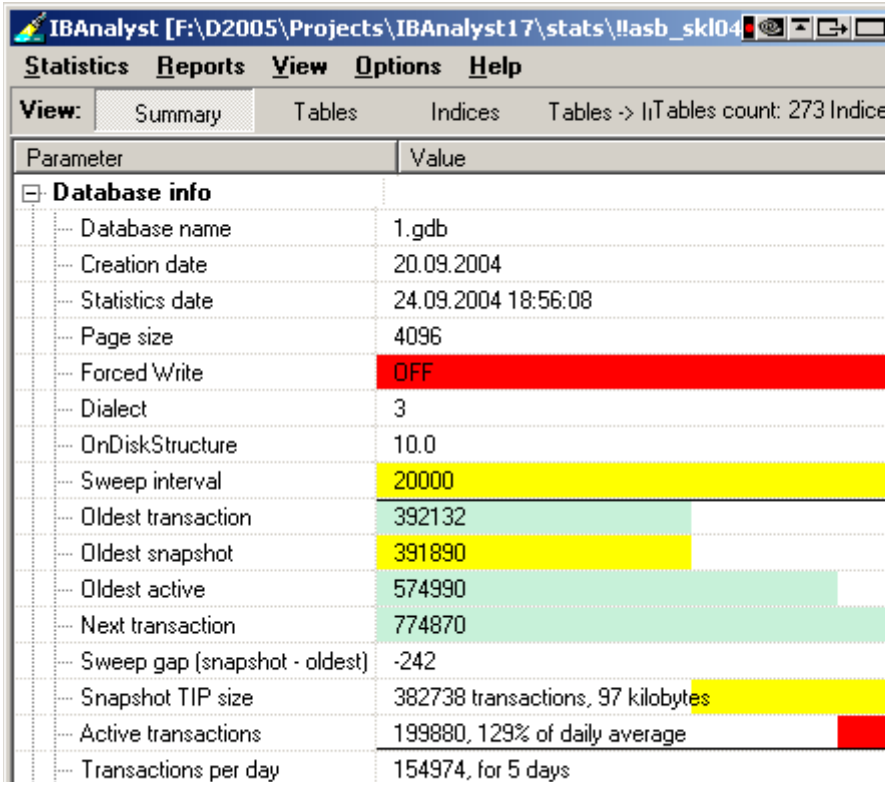
Long story, but the outcome was that IBAnalyst was created. In spite of my experience it still allows me to find very interesting things or performance issues in different databases.

Real systems have runtime performance that fluctuates like a wave. The amplitude of such 'waves' can be low or high, so you can see how performance differs from day to day (or hour by hour). Actual performance depends on many factors, including the design of the application, server configuration, transaction concurrency, version garbage in the database and so on. To find out what is happening in a database (both positive and negative aspects of performance) you should at the very least take a peek at the database statistics from time to time.

Let's take a look at the capabilities of IBAnalyst. IBAnalyst can take statistics from gstat or the Services API and compile them into a report giving you complete information about the database, its tables and indices. It has in-place warnings which are available during the browsing of statistics; it also includes hint comments and recommendation reports.

The summary shown in Figure 1 provides general information about

Figure 1 *Summary of database statistics*



your database. The warnings or comments shown are based on carefully gathered knowledge obtained from a large number of real-world production databases.

Note: *All figures in this article contain gstat statistics which were taken from a real-world production database (with the permission of its owners).*

As I said before, raw database statistics look cryptic and are hard to interpret. IBAnalyst highlights any potential problems clearly in yellow or red and the detail of the problem can be read simply by placing the cursor over the relevant entry and reading the hint that is displayed.

What can we discover from the above figure? This is a dialect 3 database with a page size of 4096 bytes. Six to eight years ago developers used a default page size of 1024 bytes, but in more recent times such a small page size could lead to many performance problems. Since this database has a page size of 4k, there is no warning displayed, as this page size is okay.

Next, we can see that the Forced Write parameter is set to OFF and marked red. InterBase 4.x and 5.x by default had this parameter ON. Forced Writes itself is a write cache method: when ON it writes changed data immediately to disk, but OFF means that writes will be stored for unknown time by the operating system in its file cache. InterBase 6 creates databases with Forced Writes OFF.

Why is this marked in red on the IBAnalyst report? The answer is simple – using asynchronous writes can cause database corruption in cases of power, OS or server failure.

Tip: *It is interesting that modern HDD interfaces (ATA, SATA, SCSI) do not show any major difference in performance with Forced Write set On or Off*[1] *.*

Next on the report is the mysterious "**sweep interval**". If positive, it sets the size of the gap between the oldest[2] and oldest snapshot transaction, at which the engine is alerted to the need to start an automatic garbage collection. On some systems, hitting this threshold will cause a "sudden performance loss" effect, and as a result it is sometimes recommended that the sweep interval be set to 0 (disabling automatic sweeping entirely). Here, the sweep interval is marked yellow, because the value of the **sweep gap** is negative, which it can be in InterBase 6.0, Firebird and Yaffil statistics but not in InterBase 7.x. When the value of the **sweep gap** is greater than the sweep interval (if the sweep interval is not 0) the report entry for the **sweep interval** will be marked red with an appropriate hint.

We will examine the next 8 rows as a group, as they all display aspects of the transaction state of the database:

- The **Oldest transaction** is the oldest non-committed transaction. Any lower transaction numbers are for committed transactions, and no record versions are available for such transactions. Transaction numbers higher than the oldest transaction are for transactions that can be in any state. This is also called the "oldest interesting transaction", because it freezes when a transaction is ended with rollback, and server can not undo its changes at that moment.

- The **Oldest snapshot** – the oldest active (i.e., not yet committed) transaction that existed at the start of the transaction that is currently the **Oldest Active transaction**. Indicates lowest snapshot transaction number that is interested in record versions.

- The **Oldest active**[3] – the oldest currently active transaction .

- The **Next transaction** – the transaction number that will be assigned to new transaction

- Active transactions – IBAnalyst will give a warning if the **oldest active transaction** number is 30% lower than the daily transactions count. The statistics do not tell if there any other active transactions between **oldest active** and **next transaction**, but there can be such transactions. Usually, if the oldest active gets stuck, there are two possible causes: a) that some transaction is active for a long time or b) the application design allows transactions to run for a long time. Both causes prevent garbage collection and consume server resources.

- Transactions per day – this is calculated from **Next transaction**, divided by the number of days passed since the creation of the database to the point where the statistics are retrieved. This can be correct only for production databases, or for databases that are periodically restored from backup, causing transaction numbering to be reset.

As you have already learned, if there any warnings, they are shown as colored lines, with clear, descriptive hints on how to fix or prevent the problem.

It should be noted that database statistics are not always useful. Statistics that are gathered during work and housekeeping operations can be meaningless.

Do not gather statistics if you:

- Just restored your database
- Performed backup (gbak –b db.gdb) without the –g switch
- Recently performed a manual sweep (gfix –sweep)

Statistics you get on such occasions will be practically useless. It is also correct that during normal work there can be times where database is in perfect state, for example, when applications make less database load than usual (users are at lunch or it's a quiet time in the business day).

1- InterBase 7.5 and Firebird 1.5 have special features that can periodically flush unsaved pages if Forced Writes is Off.

2 - Oldest transaction is the same Oldest interesting transaction, mentioned everywhere. Gstat output does not show this transaction as "interesting".

3 - Really it is the oldest transaction that was active when the oldest transaction currently active started. This is because only new transaction start moves "Oldest active" forward. In the production systems with regular transactions it can be considered as currently oldest active transaction.

How to seize the moment when there is something wrong with the database?

Your applications can be designed so well that they will always work with transactions and data correctly, not making sweep gaps, not accumulating a lot of active transactions, not keeping long running snapshots and so on. Usually it does not happen (sorry, colleagues).

The most common reason is that developers test their applications running only two or three simultaneous users. When the application is then used in a production environment with fifteen or more simultaneous users, the database can behave unpredictably. Of course, multi-user mode can work okay because most multi-user conflicts can be tested with two or three concurrently running applications. However, with larger numbers of users, garbage collection problems can arise. Such potential problems can be caught if you gather database statistics at the correct moments.

Table information

Let's take look at another sample output from IBAnalyst.

The IBAnalyst Table statistics view is also very useful. It can show which tables have a lot of record versions, where a large number of updates/deletes were made, fragmented tables, with fragmentation caused by update/delete or by blobs, and so on. You can see which tables are being updated frequently, and what the table size is in megabytes. Most of these warnings are customizable.

In this database example there are several activities. First of all, yellow color in the VerLen column warns that space taken by record versions is larger than that occupied by the records themselves. This can result from updating a lot of fields in a record or by bulk deletes. See the rows in which MaxVers column is marked in blue. This shows that only one version per record is stored and consequently this is caused by bulk deletes. So, both indications can tell that this is really "bulk deletes", and number in Versions column is close to number of deleted records.

Long-living active transactions prevent garbage collection, and this is the main reason for performance degradation. For some tables there can be a lot of versions that are still "in use". The server can not decide whether they really are in use, because active transactions potentially need any of these versions. Accordingly, the server does not consider these versions as garbage, and it takes longer and longer to construct a correct record from the big chain of versions whenever a transaction happens to read it. In Figure 2 you can see two tables that have the versions count three times higher than the record count. Using this information you can also check whether the fact that your applications are updating these tables so frequently is by design, or because of coding mistake or an application design flaw.

The Index view

IIndices are used by database engine to enforce primary key, foreign key and unique constraints. They also speed up the retrieval of data. Unique indices are the best for retrieving data, but the level of benefit from non-unique indices depends on the diversity of the indexed data.

For example, look at ADDR_ADDRESS_IDX6. First of all, the index name itself tells that it was created manually. If statistics were taken by the Services API with metadata info, you can see what columns are indexed (in IBAnalyst 1.83 and greater). For the index under examination you can see that it has 34999 keys, TotalDup is 34995 and MaxDup is 25056. Both duplicate columns are marked in red. This is because there are only 4 unique key values amongst all the keys in this index, as can be seen from the Uniques column. Furthermore, the greatest duplicate chain (key pointing to records with the same column value) is 25056 – i.e. almost all keys store one of four unique values. As a result, this index could:

- Reduce the speed of the restore process. Okay, thirty-five thousand keys is not a big deal for modern databases and hardware, but the impact should be noted anyway.

Figure 2 *Table statistics*

IBAnalyst [F:\D2005\Projects\IBAnalyst17\stats\!!asb_skl0425_11-30.txt]

Statistics Reports View Options Help

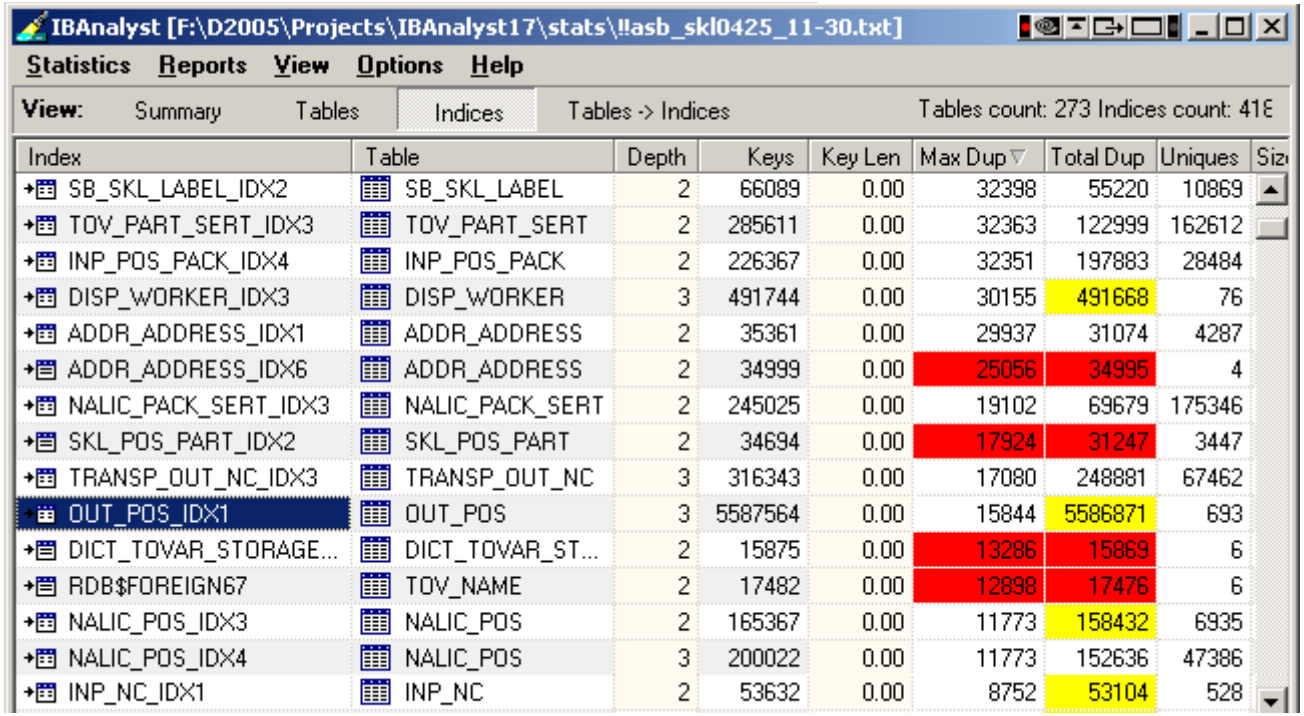
View: Summary | Tables | Indices | Tables -> Indices — Tables cour Indices count: 418

Table	Rec...	RecLength	VerLen	Versions	Max Vers	Data Pages	Size, ...	Slots	Avg fill%	RealFill
INP_MOVE_PACK	28483	62.50	15.98	168	2	732	2.86	732	76	76
PRINT_QUEUE_TMP	25186	102.72	0.00	0	0	900	3.52	900	82	82
UO_HISTORY_PLACE	24561	16.98	0.00	0	0	351	1.37	351	58	58
SKL_POS	23385	14.47	53.44	25970	12	1044	4.08	1229	61	60
SKL_POS_PART	19238	6.71	31.20	33107	5	1093	4.27	1093	49	46
SB_SKL_PART	18895	4.63	53.12	17237	1	440	1.72	675	90	90
STAT_QUICK	17649	10.44	13.32	85062	246	809	3.16	809	93	93
STAT_QUICK_WORK	17613	9.22	9.87	2076	7	218	0.85	218	59	58
DICT_TOVAR_LIST	17502	79.97	45.10	100	3	523	2.04	523	80	80
TOV_NAME	17481	89.58	50.24	71	1	565	2.21	565	81	81
SB_SKL_POS	17113	5.31	58.75	15589	1	419	1.64	711	92	91

• Slow down garbage collection. Indices with a low count of unique values can impede garbage collection by up to ten times in comparison with a completely unique index. This problem has been solved in InterBase 7.1/7.5 and Firebird 2.0.

• Produce unnecessary page reads when the optimizer reads the index. It depends on the value being searched in a particular query - searching by an index that has a larger value for MaxDup will be slower. Searching by value that has fewer duplicate values will be faster, but only you know what data is stored in that indexed column.

That is why IBAnalyst draws your attention to such indices, marking them red and yellow, and including them in the Recommendations report. Unfortunately most of the "bad" indices are automatically created to enforce foreign-key constraints. In some cases this problem can be solved by preventing, using triggers, deletes or updates of primary key in lookup tables. But if it is not possible to implement such changes, IBAnalyst will show you "bad" indices on Foreign Keys every time you view statistics.



Index	Table	Depth	Keys	Key Len	Max Dup	Total Dup	Uniques	Siz
SB_SKL_LABEL_IDX2	SB_SKL_LABEL	2	66089	0.00	32398	55220	10869	
TOV_PART_SERT_IDX3	TOV_PART_SERT	2	285611	0.00	32363	122999	162612	
INP_POS_PACK_IDX4	INP_POS_PACK	2	226367	0.00	32351	197883	28484	
DISP_WORKER_IDX3	DISP_WORKER	3	491744	0.00	30155	491668	76	
ADDR_ADDRESS_IDX1	ADDR_ADDRESS	2	35361	0.00	29937	31074	4287	
ADDR_ADDRESS_IDX6	ADDR_ADDRESS	2	34999	0.00	25056	34995	4	
NALIC_PACK_SERT_IDX3	NALIC_PACK_SERT	2	245025	0.00	19102	69679	175346	
SKL_POS_PART_IDX2	SKL_POS_PART	2	34694	0.00	17924	31247	3447	
TRANSP_OUT_NC_IDX3	TRANSP_OUT_NC	3	316343	0.00	17080	248881	67462	
OUT_POS_IDX1	OUT_POS	3	5587564	0.00	15844	5586871	693	
DICT_TOVAR_STORAGE...	DICT_TOVAR_ST...	2	15875	0.00	13286	15869	6	
RDB$FOREIGN67	TOV_NAME	2	17482	0.00	12898	17476	6	
NALIC_POS_IDX3	NALIC_POS	2	165367	0.00	11773	158432	6935	
NALIC_POS_IDX4	NALIC_POS	3	200022	0.00	11773	152636	47386	
INP_NC_IDX1	INP_NC	2	53632	0.00	8752	53104	528	

Reports

There is no need to look through the entire report each time, spotting cell color and reading hints for new warnings. More direct and detailed information can be had by using the Recommendations feature of IBAnalyst. Just load the statistics and go to the Reports/View Recommendations menu. This report provides a step-by-step analysis, including more detailed descriptive warnings about forced writes, sweep interval, database activity, transaction state, database page size, sweeping, transaction inventory pages, fragmented tables, tables with lot of record versions, massive deletes/updates, deep indices, optimizer-unfriendly indices, useless indices and even empty tables. All of this information and the accompanying suggestions are dynamically created based on the statistics being loaded.

As an example of the report output, let's have a look at a report generated for the database statistics you saw earlier in this article: "Overall size of transaction inventory pages (TIP) is big - 94 kilobytes or 23 pages. Read_committed transaction uses global TIP, but snapshot transactions make own copies of TIP in memory. Big TIP size can slowdown performance. Try to run sweep manually (gfix -sweep) to decrease TIP size. "

Here is another quote from table/indices part of report: "Versioned tables count: 8. Large amount of record versions usually slowdown performance. If there are a lot of record versions in table, than garbage collection does not work, or records are not being read by any select statement. You can try select count(*) on that tables to enforce garbage collection, but this can take long time (if there are lot of versions and non-unique indices exist) and can be unsuccessful if there is at least one transaction interested in these versions. Here is the list of tables with version/record ratio greater than 3 :

Table	Records	Versions	Rec/Vers	size
CLIENTS_PR	3388	10944	92%	
DICT_PRICE	30	1992	45%	
DOCS	9	2225	64%	
N_PART	13835	72594	83%	
REGISTR_NC	241	4085	56%	
SKL_NC	1640	7736	170%	
STAT_QUICK	17649	85062	110%	
UO_LOCK	283	8490	144%	

Summary

IBAnalyst is an invaluable tool that assists a user in performing detailed analysis of Firebird or InterBase database statistics and identifying possible problems with a database in terms of performance, maintenance and how an application interacts with the database. It takes cryptic database statistics and displays them in an easy-to-understand, graphical manner and will automatically make sensible suggestions about improving database performance and easing database maintenance.

Readers feedback

We received a lot of feedback emails for article "Working with temporary tables in InterBase 7.5" which was published in issue 1. The one of them is impressed me and with permission of its respective owner I'd like to publish it.

Alexey Kovyazin, Chief Editor

One thing I'd like to ask you to change is re temp tables. I suggest you even create another myth box for it. It is the sentence

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

This myth does not want to die. Temporary tables (TT) are not a means for underskilled DB kids, who cannot write any complex SQL statement. They are *the* means of dealing with data that is *structurally* dynamic, but still needs to be processed like data with a fixed structure. So all OLAP systems based on RDBMS are heavily dependent on this feature - or, if it is not present, it requires a whole lot of unnecessary and complicated workarounds. I'm talking out of experience.

Then, there are situations where the optimizer simply loses the plot because of the complexity of a statement. If developers have a fall-back method to reduce complexity in those cases, that's an advantage. Much better than asking developers to supply their own query plans.

Also, 'serious' RDBMS like Informix had them at least 15 years ago, when MS's database expertise did not go further than MS Access. Certainly those MS database developers who need TTs to be able to do their job would not have managed to deal with an Informix server if complexity was their main problem.

The two preceding paragraphs were about local temp tables. There are also global temporary tables (GTTs). A point in favour of GTTs is that one can give users their own workspace within a database without having to setup some clumsy administration for it. No user id scattered around in tables where they don't belong, no explicit cleanup, no demanding role management. Just setup tables as temp, and from then on it is transparent to users/applications that the data inside is user/session-specific. Web applications would be a good example.

The argument reminds me a bit of MySQL reasoning when it comes to features their 'RDBMS' does/did not have. Transactions / Foreign Keys / Triggers etc were all bad and unnecessary because they did not have them (officially: they slow down the whole system and introduce dependencies). Of course they do. To call a flat file system a RDBMS is obviously good for marketing. Now they are putting in all those essential database features which were declared crap by them not long ago. And you can see already that their marketing now tells us how important those features are. I bet we won't see MySQL benchmarks for a while ;-) .

We should not make a similar mistake. Temporary tables are important if the nature of a system is dynamic, either re user/session data isolation, or re data where the structure is unknown in advance, but needs to be processed like DB data with a fixed structure. That Firebird does not have them is plainly a lack of an important feature, in the same category as cross DB operations (only through qli, which means 'unusable'). Both features could make Firebird much more suitable as the basis for OLAP systems, an area where Firebird is lacking considerably.

Well, to be fair, Firebird developers are working on both topics.

Volker Rehn
volker.rehn@bigpond.com