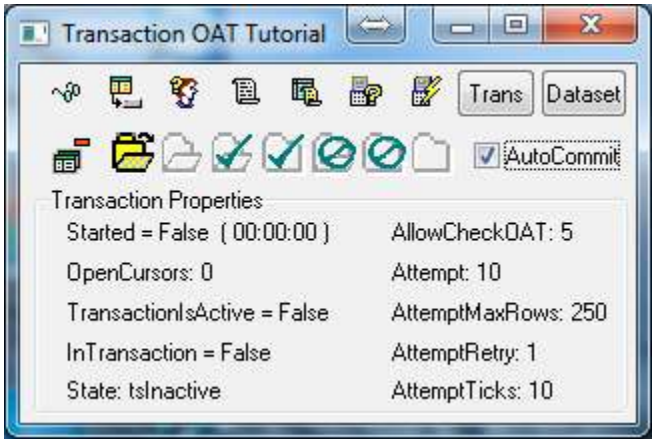
Optimizing Queries and Transactions

This is a paper based on the talk given at the Firebird Conference 2016 that includes substantial information not mentioned in the actual presentation.

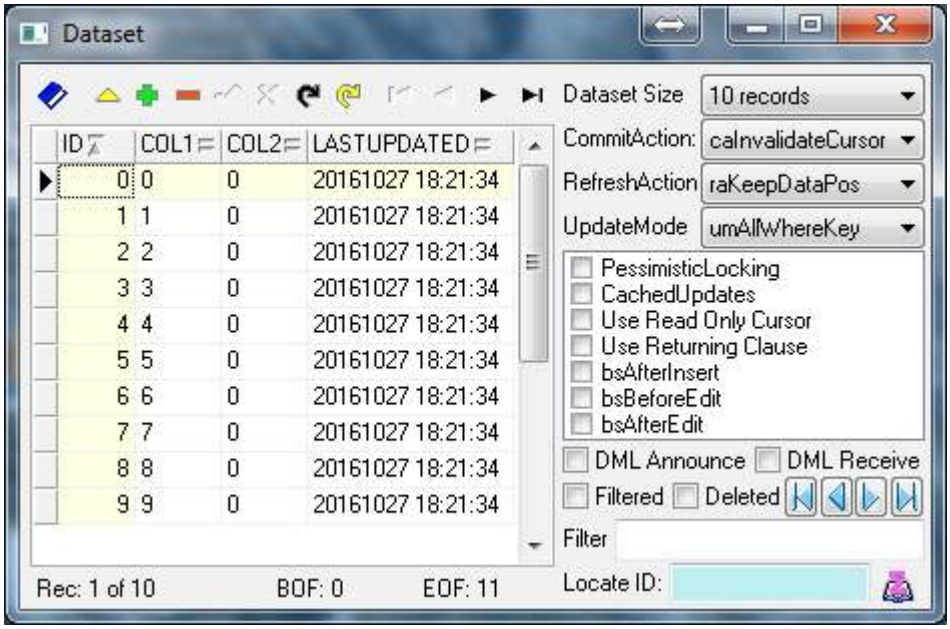
Tutorial application

This paper references a fairly simple tutorial application created specially to show several points of interest. This application is included with IB Objects following Oct 12th, 2016. The TransOAT project is found in the ..\ibo5\tutorials\TransactionOAT folder.

Here are screen shots of the application's forms:



Above is the main form that shows the status of the application's transaction. Multiple dataset forms can be open at a time by clicking the ***Open Dataset*** button.



Above is the dataset form that allows you to experiment with various dataset properties.

Tutorial database

The database for this tutorial is very simple with two generators and one table with a trigger and a stored procedure to do the initial load of its test data:

```
CREATE GENERATOR GEN_TESTDATA_ID;
CREATE GENERATOR GEN_TESTDATA_CHANGE_ID;
SET GENERATOR GEN_TESTDATA_ID TO 99999;

CREATE TABLE TESTDATA
( ID          INTEGER         NOT NULL
, COL1        VARCHAR( 10 )   NOT NULL
, COL2        VARCHAR( 10 )
, LASTUPDATED TIMESTAMP
, CHANGE_ID   INTEGER  DEFAULT 0 NOT NULL
, CONSTRAINT PK PRIMARY KEY ( ID ));

CREATE TRIGGER TEST_TRIG FOR TESTDATA
BEFORE INSERT OR UPDATE
AS
BEGIN
  IF ( NEW.ID IS NULL ) THEN
    NEW.ID = GEN_ID( GEN_TESTDATA_ID, 1 );
  NEW.CHANGE_ID = GEN_ID( GEN_TESTDATA_CHANGE_ID, 1 );
  NEW.LASTUPDATED = CURRENT_TIMESTAMP;
END;

CREATE PROCEDURE LOAD_DATA
AS
DECLARE VARIABLE TMP1 INTEGER;
DECLARE VARIABLE TMP2 INTEGER;
DECLARE VARIABLE TMP3 INTEGER;
BEGIN
  TMP1 = 0;
  TMP2 = 0;
  TMP3 = 0;
  WHILE ( TMP1 < 100000 ) DO
  BEGIN
    INSERT INTO TESTDATA( ID, COL1, COL2 )
      VALUES( :TMP1, :TMP2, :TMP3 );
    TMP1 = TMP1 + 1;
    TMP2 = TMP2 + 1;
    IF ( :TMP2 = 10 ) THEN
    BEGIN
      TMP2 = 0;
      TMP3 = TMP3 + 1;
    END
  END
  EXIT;
END;
```

The ***TransOAT.ddl*** script is found in the ***ibo5\samples\data*** folder. This is also where the database itself will be created. When the application is ran the first time, the database will be automatically created and the LOAD_DATA procedure will populate TESTDATA.

Part 1: How fetching query results impacts transactions

A transaction is required in order to submit and return the results of a query. When a query statement that can return multiple rows is executed on the server it must open what is called a "cursor" so that it can span the individual records and deliver them to the client. When the query executes and is fetched, the transaction tells it what the record's isolation level is. You might want a snapshot view of the data or you might want to see whatever the current, and possibly changing, committed state of the data is.

So, a transaction handle is required in order to maintain a server cursor. Therefore, the query itself requires a transaction handle to be acquired and held for as long as that query is opened and it remains with some records yet to be fetched. Once all of the query's records have been fetched, the server cursor used to fetch the records can be released and the transaction used to deliver the server cursor can also be ended via commit or rollback. This doesn't require that your dataset be closed since it is maintaining a buffer of those records that were fetched. This issue merely pertains to how server resources are used.

Note: This issue of server cursors requiring transactions is the reason why the BDE would sometimes seem to lockup the application. If the BDE needed to end a transaction it would attempt to fetch all of the records of the query in order to do so. If that query was very large then this would be like having your user walk into a booby-trap. Avoiding this problem imposed a rather annoying limitation on what your application could safely do.

So, if a query returns a large amount of records, this can cause a transaction handle to be allocated for a long period of time. If the dataset does not fetch all of the records then the cursor would remain open and the transaction could be opened up indefinitely. Thus, the server would have to allocate resources to maintain this transaction. This could seriously degrade your server's performance if your application has several of these types of queries or if you have several instances of the application open that do this. The fewer transactions you have open at a time the better. And, it is especially important that a transaction remain open only for a short time if possible. The longer a transaction remains open the more resources it requires on the server to maintain it. It also can prevent garbage collection from taking place, which adds to the overhead of the server.

One of the internal statistics the server keeps track of is called the Oldest Active Transaction. That is where the OAT abbreviation comes from in this tutorial. You want this OAT number to remain as close to the newest transaction possible. The more transactions exist between the newest transaction and the oldest active transaction, the more overhead there is for the server to keep everything accurate and tidy.

So, if a transaction gets started and is then stuck open in your application, this gap between the newest transaction and the oldest active transaction will grow wider and wider as other transactions come and go. Eventually, the gap increases to a point where the server starts to have degraded performance because there is such a wide gap of transactions to keep sorted out. This tutorial will focus on how to avoid this happening.

IBO optimizes transactions for queries large and small

The tutorial application allows you to simulate a dataset with queries of various numbers of records to be returned: 10, 100, 1000, 10000 and 100000 records. It will let you adjust the various properties within IB Objects that are intended to help you avoid having long running transactions while working efficiently with large datasets. I will demonstrate various combinations of these properties to showcase the way IBO deals with each case.

When a query is opened in IBO a transaction handle is automatically started in order for records to be fetched. So, when you first open the dataset by clicking on the button bar on the main form to open the dataset, you should notice "Started = True" in the Transaction Properties section. It also shows how long in seconds the transaction has been started.

Transaction isolation matters

If a query has read committed isolation IBO can automatically close that transaction because reopening a new transaction later delivers the exact same visibility of the records as it had originally. However, if the transaction's isolation is set to be a snapshot isolation, IBO will not have the ability to automatically close the transaction because this would take away the consistent snapshot view of the database you have requested. Thus, this tutorial application is showing the read committed transaction isolation only. **You must explicitly control your transactions when you use a snapshot isolation!**

Dataset Size = 10 records

With the dataset configured to fetch 10 records, all of its records are returned because the grid shows more than 10 records. Because of this, when 5 seconds has passed, the transaction will be automatically closed by IBO. This is a result of the ***AllowCheckOAT*** setting being set to 5. This means as soon as a transaction has been started for 5 seconds it will begin to see if that transaction can be safely closed. In this case, since the query has fetched all of its records, IBO was able to go ahead and close the transaction.

Dataset Size = 100 records

Now click on the 100 ***Dataset Size*** radio group option and the dataset will be reconfigured to refresh with 100 records which will require a new transaction to be allocated. After refreshing it will only have part of the records of the dataset fetched because the grid only requires a few records to be painted. This can be confirmed by it showing "***Rec 1 of 11***+" right below the grid. It says 11+ because there are currently 11 records in the buffer but the dataset's query hasn't fetched all of the records from the server.

The remaining records of the dataset will be fetched automatically due to the ***Attempt*** transaction timeout option being set to 10 seconds and the ***AttemptMaxRows*** option being set to 250 records. When this happens "***Rec 1 of 100***" indicates all of the records were brought into the buffer even though the user didn't scroll to the end of the dataset. This means when the transaction gets 10 seconds old IBO will begin making attempts to see if it can create the conditions necessary to close the transaction. In this tutorial's case, the dataset will have additional records fetched into its buffer up to the maximum of 250 records. This happens via idle CPU cycles without the user knowing that it is happening.

Dataset Size = 1,000 records (natural order)

Now click on the 1,000 option. This will put the dataset size beyond the number of records it is configured to attempt to fetch in. Thus, we will be left with a dataset size that is beyond the configured range that automatic fetches can reach. This is a dataset that we are stuck with. This requires us to explore other possible options to keep the transaction from being stuck. Selecting this option sets the query to the natural order.

Another way IBO can free up a transaction is to ***Refresh*** the dataset that it cannot fetch all of the records for. However, you can't be in an edit state and a new transaction handle will be acquired and there will be the overhead of fetching the records all over again. But, it allows the old transaction to go away and be replaced with a newer transaction. Thus, it will prevent the transaction from getting too old and stuffing things up on the server.

This even works on several datasets at a time because the ***Refresh*** is broken up into two phases. All datasets are closed and the transaction handle is released and then the datasets have their ***Refresh*** completed with the new transaction. This can be verified by opening a second dataset form and setting them both to this dataset size using the natural order.

This ***Refresh*** can cause a slight flicker in the user controls that the user might notice and it will also create a repeating cycle of refreshes. You will want to set the duration of this to a higher amount than what this tutorial application is demonstrating. You want to keep this kind of overhead to a minimum and prevent the users from seeing too frequent of flickering as their datasets are refreshed. You will have to choose how to balance it out.

You may have noticed IBO keeps the record pointer on the same record when performing a ***Refresh***. This is because the dataset is configured to the ***RefreshAction*** of ***raKeepDataPos*** or ***raKeepRowNum***. If you move this over to ***raOpen*** then the refresh may not take place and the transaction could be stuck open. However, if you have ***raOpen*** but ***KeyLinks*** are also configured then it will go ahead and do a ***RefreshKeys***.

The concern is because a user will not want their record position in the dataset moved unexpectedly to a different record or to a different row number. So long as the dataset is configured to keep the record pointer on the same record or row when refreshing, it will go ahead and use refresh as a mechanism to free up the transaction handle. Otherwise, the dataset is stuck open and the transaction will remain allocated until all of its records are fetched or until at least that dataset is closed.

You can open up a second dataset form and delete a record or two and watch what happens in the other window when the ***Refresh*** happens in order to emulate how it will behave in a multi-user environment where the committed view of the data is subject to change. This will show you that it is much safer to use ***raKeepDataPos*** than to use ***raKeepRowNum***. You must have a valid ***KeyLinks*** setting in your query for the bookmark to be made up of relevant data so that this will work. If you don't have valid ***KeyLinks*** for your dataset then it will use an arbitrary counter for the bookmark and behave much as if it is just keeping the same record pointer number.

Dataset Size = 1,000 records (sorted order via OrderingLinks)

Please click on the ***ID*** column header on the grid and sort the results by the ID column to observe the characteristics of a dataset this size that is sorted. Sorting the result set enables the cursor to be closed so that the transaction can also be released. Thus, the need for the afore mentioned refresh cycle is eliminated. The cursor will simply be revived when the time comes that more records are needed again. It uses the sort criteria in order to efficiently revive the cursor where it previously had left off.

This will all happen quietly in the background, so far as the user is concerned. They will simply see their full dataset and not be aware that there have been potentially multiple cursors involved in delivering it to them. If you watch carefully in the SQL trace monitor you can catch it syncing up to the previous record stream because when reviving a new cursor it starts with the last record fetched in and fetches it again with all of the records that follow after it in the sort order.

Thus, there is potential overlap between the cursors and there will always be overlap of at least one record. But, this is okay because when the cursors are syncing up I turn on a check to make sure that no duplicate records are received into the buffer. So, those first few overlapping fetches are simply ignored because they are already in the buffers.

I do it this way because the dataset may not always be sorted by values that are unique for each record of the dataset. In the future I will enhance IBO to detect if the sort criteria is unique or not and if so then I can make things a little bit more efficient for that case. As it is now, there will always be only 1 overlapping fetch and this is easily afforded.

But, until I make that improvement, I simply do what needs to be done in case the sort criteria allows many records with duplicate values. The important thing is that no records are overlooked and it is really inexpensive to scan my buffers for a raw key value to avoid having duplicate records.

Please follow these steps to see the details of how this works:
Select 1000 record dataset size option.
Click on the ID column header to sort the dataset by ID.
Observe how it says "Rec: 1 of 11+" after it refreshes.
Then, watch the transaction get 10 seconds old and attempt to fetch to the end.
You will end up seeing it show something like: "Rec: 1 of 250+" give or take.
Now, after it fails to reach the end of the record stream, the cursor is killed.
This is why the transaction remains unallocated and the time remains at 00:00:00.
Now you can scroll the grid down carefully until it reaches where it was truncated.
Use Page-Down and the down arrow and get to where you are sitting on the last row.
It will say ***"Rec: 250 of 250+"*** when you are right on the cusp of reviving the cursor.
Then, hit the down arrow once and bump it over the line and a record appears as normal.
However, you should notice that the transaction goes active again due to the new cursor.
The transaction will timeout and kill the cursor again and revive it again as needed.
Please notice the records have no gaps in the sequence so it is effectively one dataset.
It will pull another 250 records into the buffer before it is killed off a second time.

Dataset Size = 10,000 records (unsorted and sorted)

Pushing the dataset size up to 10,000 unsorted records puts us into a dataset size that could become a concern for performing a refresh to allow the transaction handle to be released. If the dataset is sorted then IBO will kill and revive the cursor as necessary, but you may want more efficient refreshes anyway. So, for natural order queries, we need to explore some additional parameters that can be used when dealing with larger datasets.

IBO gives you the ability to submit a query in such a way that it uses an underlying query of just the key values only and then it will bring in single records only when the other columns' values are required. This is called ***Vertical Refinement*** and it is configured simply by setting the query's ***FetchWholeRows*** property to false and IBO does the rest.

So, instead of submitting a single query like this:

```
select t.*
from testdata t
WHERE t.ID < 10000
ORDER BY t.ID ASC
```

The request gets broken up into two queries:

```
SELECT t.ID
from testdata t
WHERE t.ID < 10000
ORDER BY t.ID ASC
```

and

```
select t.*
from testdata t
WHERE t.ID = ?/* BIND_0 */
```

This allows dataset rows to be fetched very quickly because it is bringing in row keys only. Then, as needed, the individual whole records will be brought in based upon the key values already fetched in. This behavior can be verified by looking at the SQL trace monitor. I call this ***Vertical Refinement*** because the dataset is split vertically.

This makes your ***Refresh*** of the dataset far more efficient when you have scrolled somewhere deep inside of the 10,000 records. You could, for example, do a ***Locate()*** by typing 5555 into the ***Locate ID*** edit box and hitting ENTER to make it do the search. It should immediately jump to that position in the dataset. Then, the automatic ***Refresh*** to free up the transaction handle involves mostly keys only instead of the whole dataset.

The ***Locate()*** operation is also very quick because only the keys of the rows are fetched and not the individual records in between when jumping to the record found. IBO avoids pulling in the individual rows because it submits a query to the server to return the key value of the record that matches the criteria. With the key of the matching record IBO can

do a fast search in the buffer's list of keys for the key value only. Thus, it jumps very quickly to the record of interest and then only the records that appear in the grid are fetched. Also, if the record's key hasn't been fetched yet, the cursor that brings in the keys is advanced until the key is fetched. Then, the individual whole row of the matching record is fetched and the ***Locate()*** is confirmed to be correct with the whole row's data.

So, if you are using a rather random and sporadic navigation pattern in a relatively large dataset that has a lot of columns, this method of structuring your dataset would be beneficial. But, for example, if you are running a report where you are certain all the records of the dataset will be accessed, you won't want to use a dataset with vertical refinement. It is much more efficient, in terms of network packets, to have the server send the whole rows because it can bundle many of them together in a single network data packet. Using Vertical Refinement imposes the down-side of requiring an individual network request for each individual record that ends up being needed.

So, for example, a query like this one in the tutorial application, that only has 4 columns, doesn't reap a significant improvement, if any, from vertical refinement. This feature is more appropriate for a query that has several columns with random navigation through spread out ranges of records that cover a relatively small percentage of the whole dataset.

Another aspect of configuring your dataset this way is once the individual records are fetched they remain associated to their key value and are cached. So, when a query is refreshed, the individual records already fetched are preserved. As each key is brought back from the server it checks to see if it has its record's whole value stored in the cache. So, when you call ***RefreshKeys***, you won't lose the individual records. This is the same system that keeps track of cached updates and cached blobs as well. So, it is possible to refresh a dataset and still keep all of your cached updates and fetched blobs.

It is possible to see how individual record caching works by following these steps:
Click ID column header so the dataset is sorted and we get cursors that go away.
Locate ID 5555, 7777, 5555 and 7777 again and wait for the transaction to flush out.
Do a ***RefreshKeys*** on the update bar and wait for the transaction to flush out again.
Locate ID 5555 again and notice no new transaction or fetches were performed.
If you scroll to a row not yet shown in the grid, it starts a new transaction and fetches it.

For a bit of an additional test, turn on ***CachedUpdates*** and edit some records and then refresh the dataset. You can even use the button bar to toggle the dataset open and closed as well. The edited records remain viable even though the dataset was closed. In fact, you can even close the dataset and leave it closed and the updates will still be processed as they should be when the transaction is committed.

Dataset Size = 100,000+ records (sorted only)

There are some cases when queries can be put against very large tables. Under this circumstance it becomes necessary to take an entirely difference approach because even when fetching just keys it could become a noticeable delay to the user if there is a pause while it is attempting to free up the transaction handle. We need a solution that allows the user to have the query give them access to the full range of their huge dataset without having to fetch in all of the records from the beginning of it to where they have scrolled. This is where ***Horizontal Refinement*** comes into play because the result set is split up into small and easily manageable horizontal portions, depending upon where the user has navigated to. However, from their point of view, it appears as one huge dataset.

To see this in action select the 100,000 records dataset size. This will give us a dataset that is quite large but that also has horizontal dataset refinement available. You can look in the source code of this tutorial to see how this is done via the ***OrderingLinks*** property. This feature also comes automatically when using the ***TIBOTable*** component and if appropriate ascending and descending indexes are created on the ***OrderingField*** column.

This feature exists in IBO because to a limited extent the BDE also attempted to deliver this capability. So, in order to have full BDE emulation, IBO has duplicated and improved upon this feature. The improvements have gone far beyond what the BDE allowed as this tutorial will demonstrate. It extends to having complete integration with Filter, Locate(), Refresh(), Bookmarks, variations of columns indexed vs. the column or columns being searched on, and much more. In all cases possible, IBO uses the server's resources to perform the searching with minimal records actually being brought to the client. And, at the same time, transactions will not be stuck because the cursors used for fetching are able to be killed and revived on demand as a part of the horizontal dataset refinement algorithms. Thus, refreshing is not required to release a transaction handle when in this dataset mode, even when the dataset is in an edit state.

Keep a close eye on the status indicator that tells us how many records have been fetched. If you click on the ***Last*** button on the dataset navigation bar you should notice it jump immediately to the end of the dataset. But, it will still show only 11 records have been brought into the buffer. This is because it made use of one of the 4 internal cursors that are used to virtualize the entire dataset for immediate navigation throughout as follows:

```
select t.*                  select t.*
from testdata t             from testdata t
/* no where clause */       WHERE t.ID >= ?/* OLNK_ID */
ORDER BY t.ID ASC           ORDER BY t.ID ASC

select t.*                  select t.*
from testdata t             from testdata t
/* no where clause */       WHERE t.ID < ?/* OLNK_ID */
ORDER BY t.ID DESC          ORDER BY t.ID DESC
```

As you can see, there is an ascending and a descending query each with and without WHERE clause criteria for OLNK_ID. This enables us to navigate to either end of the dataset or to a point within the dataset indicated by the value of OLNK_ID.

One of the oddities of having record navigation virtualized in this manner is how the row numbers become assigned. When you are using this feature of IBO you should not rely upon the row numbers (record numbers) remaining consistent to their associated data. Each time the horizontal dataset refinement criteria is adjusted, because you jumped to the ***First*** or the ***Last*** record or a call to ***Locate()*** is made or a ***Refresh*** is performed, the row numbers are all reset. Record number 1 or 0 is always the point of primary interest of the refinement, depending on if it is in the ascending or descending orientation.

So, for example, if you do a call to ***Locate()***, the record of interest will most likely be at row number 1, if it is found. I say "most likely" because there are some cases when you are able to use partial refinement criteria. But, that goes to a level of detail beyond the scope of this paper. When another call to ***Locate()*** with different lookup criteria is made, the record number 1 would most likely be pointing to a new record entirely, if found.

You can test this out by putting the number 5000 in the ***Locate ID*** edit field and pressing ENTER. It will say ***Rec: 1 of 11***+. Then, replace 5000 with 5555 and press ENTER again and it will still say ***Rec: 1 of 11***+. In both cases it was at record number 1 but the underlying data is different because the horizontal refinement criteria was changed from 5000 to 5555. You should also notice that the user is at liberty to scroll to records before or after the one located and then, on an as-needed basis, additional records are fetched in so that the surrounding records in the dataset can appear in the grid.

You may have noticed that there are negative row numbers possible now that someone can scroll the record pointer to records previous to the one located. Or, if you call the ***Last*** method and scroll back toward the beginning of the dataset, as you navigate to previous records you will be moving in the negative direction where negative numbers are getting more and more negative as you go.

The point of origin for the record numbering scheme depends upon what horizontal refinement mode and criteria are in effect. You have the top, middle and bottom modes, depending on if you just opened the dataset, or if you located a record within the dataset or refreshed or if you scrolled directly to the end of the dataset. When in the middle zone there is also the value of the refinement criteria that the horizontal split is based upon. This value gets changed when a ***Locate()*** is performed or a ***Bookmark*** is referenced. This is how a ***Locate()*** or ***Bookmark*** can be executed and come up with an immediate answer.

So, the work of finding records in the dataset is offloaded to the server by making use of specially prepared internally maintained SELECT statements based upon your query. This feature is what enables applications using InterBase/Firebird to approach the speed of XBASE applications in dataset navigation. This is why I claim IB Objects is an excellent tool for converting BDE or XBASE applications. Most of the work of having highly efficient and fast performing datasets is taken care of for you automatically.

Part 2: How performing updates impacts transactions

When designing how your application will process its updates, you will need to choose between the two basic approaches of D**irect Updates** and C**ached Updates.** I will do my best to give you a clear picture of how each of these methods works and the strengths and disadvantages of each. This isn't an easy decision to make as there are very important consequences that are associated with each. There is a significant amount of complexity that most people don't fully notice that has to get figured out one way or another. Doing things correctly so that they are not only efficient, but most important of all, so that they are free of harmful update conflicts, is more akin to an art than a science, as you will see.

Direct Updates Introduction

Direct updates is where each update posted goes directly to the server and is immediately executed in the database. The advantage to this is you get to rely upon the excellent row level locking mechanisms of the database, if it has such. And, fortunately, InterBase and Firebird have excellent row-level locking from the very beginning.

Unlike most all of InterBase's early competitors, true row-level locking was almost unheard of. This is one reason why it was such a shame that InterBase didn't get more attention and patronage in its early times. Other databases used the database page lock as a means to help avoid update conflicts but this had very undesirable side effects. Where things that should have been available for modification, if they were sharing the same database page as something else, there could arise a needless conflict the user could do nothing about. Senseless deadlocks were a serious problem in the early days.

Because of this problem in many popular databases, the approach of direct updates hasn't really been all that popular because with direct updates, to resolve this problem, they would have to do a rollback and start their work over again. Thus, the user would lose all of their work so far with no way to recover it. The application would need a way to keep the changes made so that they could be re-applied, which is what has given rise to the cached updates approach because by design it overcomes this particular weakness.

Not only were users subject to possible conflicts, senseless or otherwise, there is also the issue of having some kind of fault with your connection to the database. If your network went down then your connection to all of your updates posted so far is lost. When the user got reconnected they would have to start their work over in a new transaction.

All during the time that a user is working with direct updates there is a transaction allocated so that all of those changes will either commit together or be rolled back as one unit of work. That's the beauty of transactions is if you suffer some kind of a loss, at least you can be certain that everything either succeeded together or that it failed together. This greatly simplifies our job as database programmers. However, this means your application is keeping a transaction open for an indefinite period of time putting a significant workload on the server to mitigate all of this potential complexity.

Another thing that is an excellent feature of InterBase and Firebird is stored procedures and triggers. This enables the database to become directly involved in ways that were unthinkable with simple flat file database systems. One of the greatest benefits of using direct updates is you can put a lot of your logic directly inside of database programming and you can access it directly as you go along making your changes.

So, each time a post is made, programming in the database can react and do things on its own and those actions will then be immediately visible to the client even before the transaction is committed. So, database programming can be a direct part of applications that use direct updates. Otherwise, the database programming will only be executed at the time the updates are all being applied because they have been cached and kept from the database purposely. This is one of the major trade-offs between the two modes.

Cached Updates Introduction

Cached Updates is where all of the user's updates are kept in a local buffer or buffers of some kind in waiting for when the user has completed all of their work and then they wish to submit the whole batch of changes together. These updates are typically stored in each individual dataset within a record-level update cache.

A major reason why the cached updates model has been given a lot of use is because it holds all of your changes and safeguards you against senseless update contentions and failed connections. It allows you to start a fresh new transaction to attempt to post all of your changes in one go of things. If for some reason there is a problem and it fails, everything posted in its attempt to apply the updates is rolled back and you are put right back to where you were prior to attempting to submit your updates. If there is success then a commit is performed and everything is kept together as a whole unit of work.

Transactions greatly simplify one of the most complex aspects of database programming. When I worked at U-Haul International there was a lot of difficult code written just to try and clean up the fall-out of failed so-called "transactions". We were using an XBASE database and true transactions were non-existent. Thankfully, those days are behind us.

Thanks to true transactions, the cached updates model becomes even easier to use because one single transaction can be used to apply all of the updates or your application can do a rollback to clean everything up as if you had never made any updates.

So, if there is a conflict in the database when applying updates, it could indicate what it was and allow just that specific item to be addressed and then submit all of the updates again with the new changes to address the problem. This cycle could be repeated as many times as necessary until the user is successful at applying their updates.

Upon a successful applying of all of the cached updates, the update buffers can be cleared out and all of the new changes will be available for the queries to refresh directly from the server. Any database programming invoked in triggers and/or stored procedures then becomes visible and the datasets can be refreshed to reflect all of the new results.

Performing Updates Introduction

There are actually a number of ways updates can be submitted to the server:

- Edits and deletes can be performed on records positioned by the server's cursor.
- Edits and deletes can use a search by key columns alone or with all others.
- Edits can use a search by key columns plus columns that have been modified.
- All updates can be a prepared statement or an immediately executed statement.
- Edits and inserts can have a RETURNING clause to get values from the server.
- Edits can synchronize with the server upon going into edit state and after posting.
- Inserts can synchronize the record inserted with the server after posting.

Using direct updates favors a different way of performing updates than the way cached updates favors. There are some important technical reasons for this I hope to demonstrate. The main difference revolves around addressing update conflicts.

We want to be made aware when an update conflict happens so that our application doesn't accidentally overwrite one user's update with another user's update. We also want our updates to be processed in the most efficient manner possible. So, let's dive in and see how each way of performing updates looks with our two approaches.

Positioned Updates

What it means when we say "positioned" is the record to be edited or deleted is what the server's cursor is currently positioned on. This way of performing edits and deletes is useful when you are scanning a unidirectional cursor with a live cursor via the FOR UPDATE clause on the SELECT statement. This clause makes the server send only a single record at a time so that it will keep track of that position in case positioned updates are desired. You would also need this if you want a live query that involves a SELECT statement with a JOIN. InterBase and Firebird are able to deliver such queries as live.

So, when using positioned updates, it becomes possible to identify the record of interest simply by using the following clause with your UPDATE or DELETE statement:

```
DELETE FROM MYTABLE WHERE CURRENT OF <<CURSOR NAME>>
```

This statement will delete the most recent record fetched from that cursor in that table.

In IBO the ***TIB_Cursor*** component uses positioned updates by default by having the ***SearchedEdits*** and ***SearchedDeletes*** properties set to false. Though this is the default, it may still make sense to use searched edits and deletes because putting a SELECT statement in the FOR UPDATE mode makes it much less efficient at fetching records.

I'm not going to say much more about this updates method because it isn't a mainstream way of handling updates with higher level buffered datasets, even though it does work with a TIB_Query component. What happens is IBO opens a cursor for the single record of interest and then allows the update to take place relative to that single record's cursor. There is substantial overhead to make a cursor for each individual record to be updated.

Searched Updates

This method of performing updates is the most straight forward because we are directly in control of the record selection criteria. And, as I will show you, we have quite a bit of flexibility to safeguard against record update contention by way of using regular column values in addition to key column values when performing them.

If you are using direct updates and relying wholly upon the database server to prevent update conflicts, it is acceptable to just use the key columns as the record selection criteria in your update statements, as long as you have ***PessimisticLocking*** set to true.

This is possible because InterBase/Firebird has true row-level locking. Once your transaction has an update or a delete posted to the server it remains locked until your transaction is committed or rolled back. If any other user attempts to lock or update that record they will receive a deadlock exception. Thus, you can be assured that until your work is completed, nobody else is going to overwrite your update.

This is what a searched edit and delete statement will look like:

```
UPDATE TESTDATA T
  SET T.ID = ?/* T.ID */
    , T.COL1 = ?/* T.COL1 */
    , T.COL2 = ?/* T.COL2 */
WHERE T.ID = ?/* OLD.ID */

DELETE FROM TESTDATA T WHERE T.ID = ?/* OLD.ID */
```

This is the most basic method of applying updates and it will get the job done. But, to be sure you are avoiding update conflicts, I highly recommend putting additional measures in place. The reason for this is, for example, a user could have opened up a form and then leave the room for a while. The data displayed on their screen is the old version of the record. In the meantime, another user could have modified that same record and committed their change. However, when the first user posts their change after returning, their update could overwrite an update that they never even knew existed.

So, even with row-level record locking on the server, there is still a weakness where users could step on one another's toes, so to speak, and wipe out each other's work.

Pessimistic Locking

As I mentioned above, we totally avoid inadvertent update conflicts by placing a lock on each record at the time it is put into edit state. In this way, you prevent other users with another instance of your application from even attempting to make a change to that record until you have finished your work. If they attempt to put that record into edit state and another user has already done so, then they will get a notification telling them that this record has been locked by another user.

Obviously, this method of preventing update conflicts requires you to be operating in the direct updates mode, as opposed to the cached updates mode. This is the only way to have the server involved in fully locking down your records when at the same time you are only using key columns as the searched update record selection criteria. This also does not mean other applications won't be able to go into their own edit state of some kind and attempt to post changes based on out of date information.

In IBO there are two properties you need to use in order have this increased level of conflict avoidance in your application. You need to set the ***PessimisticLocking*** property to true in order to have the record locked upon going into edit mode. If you also want to have that record refreshed with its most recent data from the server, you need to include the ***bsBeforeEdit*** flag in the ***BufferSynchroFlags*** property. This prevents you from editing the record with data that has become out of date.

You can observe how this works by setting the ***PessimisticLocking*** and ***bsBeforeEdit*** options and by watching what happens in the SQL monitor when a record is put into edit state. You should see a statement executed like this to acquire the record lock:

```
UPDATE TESTDATA T
  SET T.ID = T.ID
WHERE T.ID = ?/* OLD.ID */
```

And, you should also see a statement like this executed for the record level refresh:

```
select t.*
from testdata t
WHERE t.ID = ?/* BIND_0 */
```

Perhaps it appears that the UPDATE statement doesn't actually do anything because it just sets the T.ID column to its current value. But, even if the value does not change, and we really don't want it to anyway, the server still flags that record as having been updated. This update functions as a lock and is sometimes called a "dummy update".

When this lock operation is performed, your database triggers will be executed so you will want to be aware of how this will affect things. As a rule of thumb, your triggers should always look for actual changes in the data to react to. By doing this, you will avoid having this lock operation make it look like the record was actually changed.

This lock will remain in place until your transaction is committed or rolled back. So, if a user puts a lock on a record and then cancels the edit state, IBO sees if it is possible to go ahead and release the lock immediately by performing a ***Rollback*** or ***RollbackRetaining***, depending on if there are open cursors that need to be kept alive. However, if there are already other updates posted to the server in this transaction that are waiting to be committed, then I am not at liberty to release the lock because those changes would also be lost. So, in this case, the lock remains bundled into whatever work is going on.

Let's experience some conflict

If you click on the ***"Trans"*** button you can open up a new transaction form with a second dataset form for that other transaction below it. In this way, you can simulate another user and actually test how the record locking mechanisms work. You should also open up the SQL Monitor by clicking on the ***TIB_UtilityBar*** button that launches it.

Put both datasets in ***PessimisticLocking*** mode and try and edit the same record in each transaction's dataset form. You should see the following message appear when you attempt to put the second dataset in edit state:

This message is generated from the default application handler so it is a bit more friendly and informative than the standard default error message. At some point you will want to examine the ***OnUpdateError*** event handler in the ***ApplicationHandler*** as it demonstrates many useful tools for handling exceptions when performing locks and updates. More will be said of it later.

Let's look at what can still go wrong. Set ***PessimisticLocking*** to true and ***bsBeforeEdit*** to false. Also, make sure both datasets are refreshed to show the current latest committed data. Now, in the first dataset form do an update and post it. Then, you will see that you can do an update in the second dataset form and post it as well, except that it never was aware that another user had posted a change. That other user's update was never brought to their attention prior to posting their changes. Allowing a user to trample over the work of another user like this could be a problem. This is what I call an inadvertent conflict.

To cure this set the ***bsBeforeEdit*** option on each dataset form and proceed to make the same changes as was done before. You should notice when the second dataset goes into edit mode that it grabs the most recent version of that record from the server. So, the change you just made and posted in the first form is now visible in the edit mode of the second form. Thus, we have made it so that it is not possible for users to trample over each other inadvertently. As they lock the record they also get it's most current value.

While this method of conflict resolution is very thorough and secure, it comes with a rather high price tag, in terms of server resources. This is mostly because it requires potentially long-running transactions. You would need to put safeguards in your application against them being too long. IBO has mechanisms that allow you to configure timeout parameters for your transactions. You can use the ***TimeoutProps*** property to determine when they are prompted and forced if necessary.

This style of long-running transactions in order to rely fully on the server to prevent update conflicts could affect your database environment in a way that makes other operations, such as batch processes, prone to interruption. This is because the records they need access to for processing could be locked down for an indefinite period of time.

Fortunately, this update mode doesn't affect long-running reports because they start a transaction with a snapshot view of the database ignoring any new locks/changes. It is easy to take this for granted but there were competing databases that had "readers" blocking "writers" in order to deliver a stable snap-shot view. InterBase/Firebird has never had this problem because of its multi-generational record versioning capabilities.

So, using direct updates with row-level record locking can be a workable and reasonable approach to developing your application. It gives you direct access to all of the server's programming as you perform your drawn out transactions. Unless you know of a good reason not to use this approach, this is what you will want to do. Direct updates is a very easy method to implement but it can have some draw backs as well.

Optimistic Conflict Resolution

Let's take a look at another way to resolve the above mentioned problem of a user posting a change that was based on data that became out of date during the time they were working with their data. We will solve this in a way that does not require any long-standing transactions.

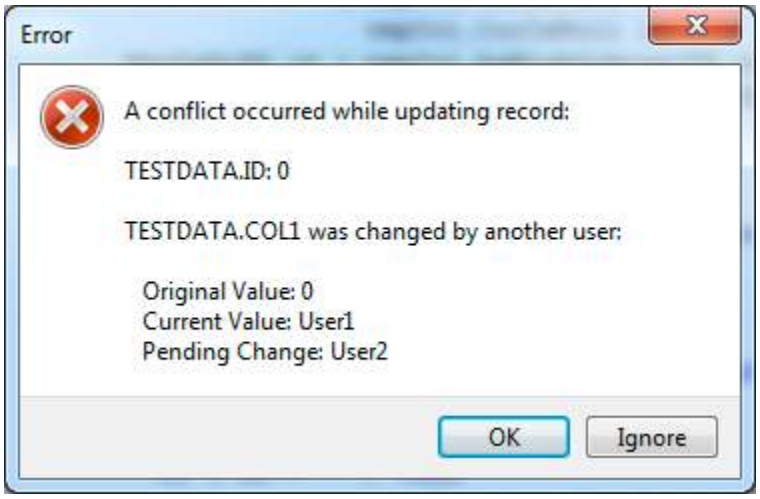
The simplest way of detecting update conflicts is to modify our record selection criteria of our UPDATE and DELETE statements so that we will only take action on a record that yet has the same original values in the database as we were working with as we performed our edits, etc. In this way, we can detect if another user has put in some changes during the time we fetched our data and then submitted our changes.

Here is an example of a searched update designed to detect update conflicts:

```
UPDATE TESTDATA T
 SET COL1 = ?/* NEW.T.COL1 */
WHERE T.ID = ?/* OLD.T.ID */
  AND T.COL1 = ?/* OLD.T.COL1 */
  AND T.COL2 = ?/* OLD.T.COL2 */
  AND T.LASTUPDATED = ?/* OLD.T.LASTUPDATED */
  AND T.CHANGE_ID = ?/* OLD.T.CHANGE_ID */
```

In this example we not only have the key column to uniquely pinpoint the record we are interested in, but we have also included the other columns of the table and plugged into them the original values we had. Therefore, if there was a change to any of those other columns' values then this update will actually fail to select and update the record and it will give a ***RowsAffected*** result of 0. Using ***UpdateMode umAllWhereAll*** does this.

Follow these steps to see how conflict detection based on Rows Affected works:
Click on the button that says ***"Trans"*** to get a second transaction form/context.
Set the ***UpdateMode*** property to ***umAllWhereAll*** on each dataset form.
Make some changes in each dataset that will conflict with the other.
A form like this should appear showing the values involved in the conflict:



This says column COL1 had a value of '0' and User2 was trying to change it to 'User2' but User1 managed to change the value to 'User1' in the meantime.

Clicking ***OK*** acknowledges the exception and the post or commit is interrupted.
Clicking ***Ignore*** pushes your change through and ignores the changes on the server.

You will want some way like this form to show the difference between what is on the server and what the original old values were. Unfortunately, the data bound controls don't show the server's current values or the old values. But, be sure to examine the tutorial app's source code and ***IB_VCL.pas*** to see how this works. The primary properties involved are ***AsString, OldAsString, CurAsString, CurIsAbsent, CurValue, etc.***

The way the ability to ignore the changes on the server works is in the ***OnUpdateError*** event the ***UpdateAction*** is set to ***uacIgnore*** rather than just ***uacRetry***. Think of it as a retry that washes out the original values updating them to the current values. This makes it so the update being attempted will be successful because when it is attempted again the ***RowsAffected*** will be 1. This makes the OLD value parameters in the WHERE clause align with the server's current values for it to successfully apply the update.

Allowing the user to ignore the new values on the server should be acceptable since the dialog presented the other value or values being wiped out. The decision to overrule those old values was an informed one. I even convert the edit into an insert if it detects the record was removed from the server. And, if a delete detects a record was already deleted, it will just mark itself as having been applied. And, if an insert detects that a conflicting record with a duplicate key has already been inserted it will give you the opportunity to simply edit that existing record and retain the current insert as a success.

UpdateMode

IBO takes care of the chore of putting together your SQL statements to apply your updates to the server. Setting the ***UpdateMode*** property has some potentially significant effects on how your application will work. It is important to understand this property.

The default mode for TIB_Query is ***umAllWhereKey*** and for the unidirectional TIB_Cursor it is ***umAllCursorPos***. We also experimented with ***umAllWhereAll*** above. Let's take a look at the other modes that are available.

umAllWhereKey	All the query's columns. Searched by keys only.
umModWhereKey	Only modified columns. Searched by keys only.
umModWhereMod	Only modified columns. Searched by keys and modified columns.
umAllWhereAll	All the query's columns. Searched by all the query's columns.
umModWhereAll	Only modified columns. Searched by all the query's columns.
umAllCursorPos	All the query's columns. References server's cursor position.
umModCursorPos	Only modified columns. References server's cursor position.

The preface of these is either umAll or umMod. What this means is it determines what columns are included in the insert or update statement. Does it put all of them in the statement or does it only include the columns that have had a new value assigned to them.

There is a tradeoff between these two styles. On the one hand, by always including all of the columns your statement remains consistent from one update to the next, regardless of what changes are taking place in the query. This allows your statement to be prepared once and then kept on hand to be used quickly and immediately for each succeeding update you are posting. This reduces the level of network chatter significantly.

However, if your query has a large number of columns and you are only updating a few values here and there at a time it can put a lot of data over the wire that doesn't actually have to be transmitted to the server. If you have only changed the values of a couple of columns you really don't want to have an update statement send 50 or a hundred column's values in order to perform the update. So, it can also backfire and cause waste too.

The suffix of these is Key, Mod, All or Pos. This tells how the record being updated is selected and acted upon. Pos means it is using the server's unidirectional cursor position and so no other record selection criteria matters. The others all refer to how the searched mode of updates works. It will either use just the keys or just the keys plus any values that have changed or it will use all of the columns. Each of these has a purposes for how data conflicts can be handled.

Even though it is the default for buffered queries, I discourage using the keys only modes. This is the default because it is the simplest to use. Users will rarely see deadlocks, key violations, conflicts, etc. because most of what one user does just overwrites what the other users have done. A professional application should do more.

The tutorial application uses a buffered query so it isn't really practical to show how to use the cursor position modes. These are really only ideal for working with unidirectional datasets. But, it will actually work with a buffered query too. What happens is it will fetch the individual record of interest in a live cursor and then apply the update to that special cursor provided just for the update. However, the overhead required to do this isn't really worth it since you can just do a searched update directly without wrapping it all inside of a cursor on the server. Therefore, I won't go into this mode any deeper.

What remains is to show the difference between using all of the columns vs. using modified columns. The principle you want to have in mind here is whether or not your column values are all independent of one another or if they are tied together as a group of interrelated values that don't really stand on their own.

For example, a person's address is composed of several columns and so if an address is going to be updated you would want all of those columns to be kept together. You wouldn't want the street location of one address and the city of another location to be mixed together. So, if someone has updated an address then you want all of the columns of the address they have worked with to be posted together as a unit. Depending on how you configure your ***UpdateMode***, you can allow changes to be mixed together or to block one another as a group.

Here is how things can be mixed together using the ***umModWhereMod*** mode:
Open up the app and click on the Trans button to get a second user transaction going.
Put each dataset's UpdateMode to ***umModWhereMod***.
In the first dataset's form change the value for COL1 in a record.
In the second dataset's form change the value for COL2 in the same record.

Both changes were allowed because this update mode works at the individual column level. Notice when the second update is posted the value from the first appears. This is what I call bleeding through. In some cases you may want this and in other cases you may not want it. Using ***umAllWhereAll*** prevents things from bleeding through.

Go ahead and give it a try making the same updates again but with the ***umAllWhereAll*** update mode and see how it reacts. You should see a dialog prompting you to either cancel what you are attempting to do or to overwrite the entire record with all of its values, including the ones you have not changed, if you decide to ignore the conflict. In any case, edits from one user's post and edits from your post will not be allowed to bleed through with one another. This should conclude the critical differences between modes.

You might wonder why there isn't a ***umAllWhereMod*** update mode. I deliberately omitted it because there is no practical or beneficial use for it. It doesn't make sense to attempt to modify column values that you have only performed partial checks for conflicts on. It would make more sense to just use the keys only mode.

Cached Updates

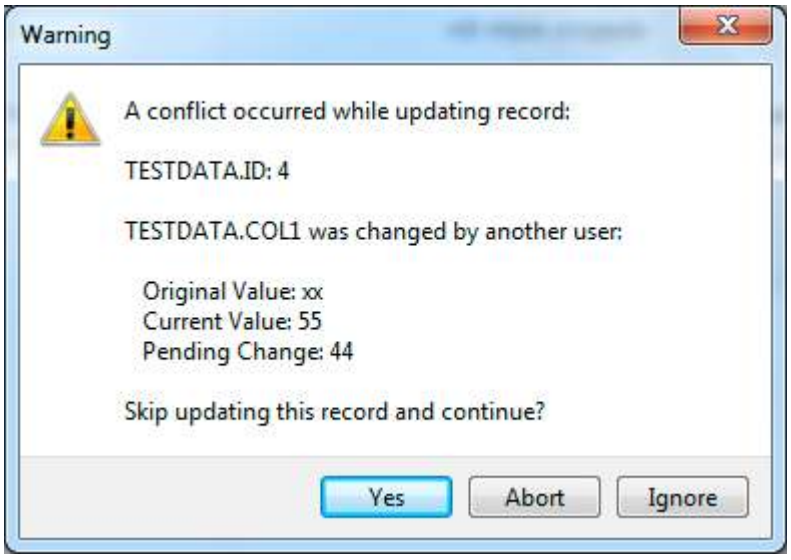
Let's have a look at how cached updates work now. With cached updates the updated and posted record isn't actually applied on the server until after all of the records of interest have already been posted. These posted update actions are stored in memory and then when you are ready they are all applied to the server together in a burst of updates.

This introduces the aspect of performing the ***ApplyUpdates*** process when you want everything to be sent to the server. Now, instead of posting an individual record as before with direct updates, we are performing a batch of multiple records. All of this takes place inside of a specially designated transaction that will either end with success and commit or it will fail and everything will be rolled back.

This is the beauty of cached updates. You only need a transaction to be held open during the time you are sending all of your updates in a single batch. Otherwise, depending on what your datasets are doing with their cursors for fetching, there is nothing that should require a transaction to stay open for very long.

IBO gives you the ability to respond to data conflicts in different ways. You can go ahead and abort or you can ignore the conflict and override it or you can skip over it and continue with the remaining items in the batch. Skipped items remain as a cached update after all of the others are committed. Let's set up some examples to have a look at.

Open a second transaction form so that we have two datasets to work with.
Turn on ***CachedUpdates*** and set the ***UpdateMode*** to ***umAllWhereAll*** in each form.
Edit the COL1 value in 3 consecutive records in the first dataset form.
Edit the COL1 value in one of the 3 records over in the second dataset with a conflict.
Apply the update in the second form to create the data conflict for the first form.
Now attempt to apply the updates in the first form and notice a dialog like this appear:



Buttons will appear allowing us to respond as they are suited to what is going on.

Here is what each button that can potentially appear in this form does:

Clicking ***Yes*** causes the item to be skipped. This allows the remaining updates to yet be committed if no other problems come up that cannot also be skipped. If ***ApplyUpdates*** is ultimately successful, skipped items are put back as they were and the user can continue to work with those items and attempt to apply them again.

Clicking ***Abort*** aborts the ***ApplyUpdates*** process and puts things back to how it was. This gives the user everything back to where they were so that they can address the problems with the whole bundle of updates together.

Clicking ***Retry*** simply attempts the same statement over again. This wouldn't be a useful option for this particular case because there isn't going to be a resolution by waiting a little while. Thus, this button does not appear here. However, as we saw earlier above, if there is a deadlock, the user might just need to wait until the lock is released. I also put into this app a dialog asking if you want to simulate an action of the other user causing the deadlock so that you can see how this works.

Clicking ***Ignore*** allows the update to go ahead and be applied, winning the conflict. This means the record being updated is refreshed with the values from the server to update its original data to the new server value. Thus, as was explained above in more detail, the OLD values were brought into alignment with the server's current values allowing the update to be successfully applied.

There is a current weakness in that a user could start an apply updates and walk away from their computer. If this dialog appears and is waiting for them to respond, this has the unfortunate consequence of holding the transaction open all during the time this form is waiting. Thus, it is highly advisable to write your own custom OnUpdateError code that ensures if a user is prompted that there is an adequate timeout mechanism in the event that the user is unable to respond in a timely manner.

DML Caching

This feature adds a very powerful tool for avoiding data conflicts. It's goal is to have your buffered datasets pro-actively updated in real-time with whatever changes are taking place in the data. It is a system of propagating messages throughout your application so that buffer synchronization can take place as efficiently as possible. Fortunately, it is a simple feature to use.

A dataset can be configured to announce and/or receive DML caching messages separately. This can be seen in the tutorial app by opening a second transaction and dataset and then in the dataset forms click on DML Announce and/or DML Receive so that the dataset forms will generate and/or listen for the appropriate messages based on what is going on in the application.

Once this is done then proceed to make edits, inserts and deletes in a dataset form configured to announce and then observe how those same changes are automatically applied to the other form that should be configured to receive. You might also notice that a little pattern of dots will surround the grid cells that are affected. This is based on the conflict management system keeping track of changes injected from an external source and attempting to draw special attention to them.

The reason this is called DML "caching" is because there are different scopes in which the messages generated can be applied. It doesn't make any sense to propagate a DML message from a transaction that has not yet committed to datasets in other transactions. They wouldn't be able to see the new changes on the server yet, not until there is a commit performed.

So, there are some messages cached up until the transaction they took place in performed a commit. Once this happens then it attempts to announce all of the messages from its cache to the other transactions in the connection. Also, before the commit takes place, the DML messages are propagated among the transaction's other datasets right away. We can see how this works by clicking on the Dataset button to get a second dataset within a transaction.

Here are some steps to take to see how this works:
Start the application fresh and click on the Trans button and then the Dataset button.
Click on the first transaction form's "StartTransaction" button bar button.
Click on the first transaction form's first dataset form's DML Announce check.
Click on the other two dataset form's DML Receive checks.
Proceed to make edits to records in the dataset configured to announce.
Notice how those changes only show up in its companion dataset form below it.
Notice how those changes are not appearing in the other transaction form's dataset form.
Now click on the first transaction form's "Commit Transaction" button bar button.
Notice how the edits are then synchronized over to the other dataset form.

This feature works with both direct updates as well as with cached updates. Take some time to play around and experiment and see how it works. For example, batch up some conflicting cached updates in dataset forms and then apply one of them. When the changes are synchronized over to the dataset that has active cached updates it will actually detect update conflicts and highlight them with a subtle red rubber-band around them. Configure all of the datasets to both send and receive and play around with it.

You can also try different UpdateMode settings to see how the locking mechanisms work in conjunction with having DML Caching configured. Fortunately, with the DML caching there will be far fewer conflicts because the data is proactively synchronized in real-time.

RETURNING clause

An excellent feature in Firebird is being able to put a RETURNING clause on its UPDATE and INSERT statements. This allows singleton updates and inserts to immediately return any new affected values from the update or insert. This makes it efficient to get values from columns affected by triggers, for example. It even becomes possible to have a primary key value provided in a trigger and given back to the client. So, this not only eliminates the additional work of fetching a record after the insert but it can also eliminate the query to acquire new key values from sequences when inserting a new record. This is also great for getting values from COMPUTED columns or other columns that the server derives.

This statement clause also helps out in other ways. I generally avoid using the ***"execute immediate"*** mode for posting dataset updates, even though there is a bit of a performance improvement to just throw a statement to the server in a single go. It is worth it to allocate a statement handle, submit a statement to be prepared, described and then to be executed if it means you get confirmation the intended record was actually affected. Otherwise, we couldn't know and this could be problematic since this is how conflict resolution is performed.

However, if you include a RETURNING clause on your statement that is executed immediately, the nature of the statement is changed to behave like a singleton SELECT statement that expects exactly 1 row to be affected. If there are 0 or 2 or more records returned by a singleton SELECT then an exception should be raised.

Therefore, it is the same logic when performing an UPDATE with a RETURNING clause. The question of having a rows affected of 1 becomes an implicit part of performing an update with a returning clause. And, if the server doesn't give an exception, I still put a safeguard to check if meaningful values were returned. If NULL is returned for all of the columns being returned then it did not actually affect the record we wanted it to and I treat this the same as if it had a rows affected result of 0.

When using the RETURNING clause with cached updates, the values brought back at the time of applying the updates will be cancelled out if there is a failure somewhere in the ***ApplyUpdates*** process and a subsequent rollback is performed. The goal is to put the user right back where they were when ***ApplyUpdates*** was first called. These values need to get thrown away because the transaction they represent also went away.

Note: Only later versions of Firebird have support for the RETURNING clause and only Firebird 3 fully supports it with positioned updates and with the execute immediate statement execution. InterBase does not support this at all, even though I have been begging them for years to support it. It is a very nice feature that makes more optimal use of bandwidth. It's a great way to optimize your application.

Conclusion

Hopefully this paper will have given you a deeper understanding and appreciation of what all goes into writing efficient and transaction friendly applications that safeguard your user's updates against conflicts in a way that keeps things convenient for the user.

There is a lot of capabilities build into IBO waiting for you to dive in and explore.

Enjoy!