The mystery of RDB$DB_KEY

By Claudio Valderrama - Copyright www.cvalde.net

If InterBase® was for years Borland's best kept secret, probably rdb$db_key is the best kept InterBase® secret, because it gives the user some extra power to access records in raw mode. Usually referred simply as db_key, it may help you to perform some magic things or improve performance but because it's an exclusive InterBase® feature, it will tie your SQL code to InterBase®. On the other hand, if you're worried about the dependency on InterBase®, you should know or remember that several built-in functions that you use in Oracle or SQL Server® are proprietary extensions, so perhaps you're already dependant on another engine without realizing that fact. InterBase® is rather poor in built-in functions, the ones provided are exclusively standard (with the exception of gen_id for generators) and the extensions are clearly separated as UDFs.

The information put here comes mainly from my experience and from some explanations I've read over the prior years, so submissions are welcome. Unfortunately, the manuals say almost nothing about rdb$db_key, so it's not a surprise several developers don't use it.

First, let me state in plain words that db_keys are glorified record positions. Relational theory says nothing about implementation details. Conceptually, a relational database knows nothing about a record number, it only works with set of records. However, it's clear the implementation must deal with these details. InterBase® makes explicit to the user such record position if the user requests this information.

In Paradox® and dBase®, using absolute record numbers was a common practice due to the desktop and single user nature of these products, although Paradox® has multi-user capabilities that makes dangerous assuming too much about absolute positions in a networked environment, because another user can drop a record that was known to exist ten minutes before. In a desktop database, a record number is a fast way to reach the desired record. Almost no translation is needed: given a record-num, the size of each record is known, so the position of the target record is easily calculated and the position to read inside the file holding the record is sent to the operating system if such information is not already in the buffer of the application. When I say “almost no translation” I'm writing from the point of view of the database. Obviously, the operating system must translate a virtual disk address in its file system and pass the raw address to the IDE/SCSI controller than in turn passes such address to the firmware of the hard disk… and you still think it's easy to identify the culprit when a read error happens?

The only official information about db_key that I could find in the InterBase® 6 documentation is in the Language Reference and it's in the chapter System tables and views, where the fields of rdb$relations are detailed:

RDB$DBKEY_LENGTH SMALLINT Length of the database key. Values are:

- tables: 8.
- For views: 8 times the number of tables referenced in the view definition.

Do not modify the value of this column.

Not too much, after all. But you know that for tables, db_key uses 8 bytes and for a view, if such view joins three tables, then its db_key uses 24 bytes. This is important if you are working with stored procedures and want to keep db_key in a variable. You must use a char data type of the correct

length. How does a db_key look in a query? Let's use isql.exe, the command-line utility with an arbitrary table:

```
SQL> select rdb$db_key from correlativo;
DB_KEY
================
0000008600000001
0000008600000002
0000008600000003
0000008600000004
0000008600000005
```

So, at first glance, there would be some error in the documentation. To clear the confusion, we will use another table:

```
SQL> select rdb$db_key from images;
DB_KEY
================
000000B600000002
000000B600000004
000000B600000006
000000B600000008
000000B60000000A
```

Then you can realize that the trick is only visual: by default, db_keys are shown as hex values, so you need two hex digits to represent a byte and you have 16 hex digits, 8 bytes. The documentation is correct. One implication is that the largest position of a record is (using unsigned numbers) is 2568=1616, namely, 264 ~ 1.84E19. Knowing that the maximum InterBase® database in theory is 32 TB=245 and that one single table can eat these 32 TB, if at first glance you naively think that each db_key value where mapped to a physical location in secondary storage (you will see that this is not true), the range suffices.

However, given that databases must be split into files of 2 GB or 4 GB (depending on file system), the calculation is not so simple. In some sense, InterBase® is using 64-bit internal addressing, but not at the operating system level: it relies on 32-bit file systems still.

In some tables, -each db_key is one more than the previous while in other tables, each db_key is two more than the previous, for example, so this may be due to the record's length. See the two examples above. While personal digressions may be funny, they may convey wrong ideas to the readers, so I decided to ask an expert and here's the response from David Schnepper:

```
 I think you have a pretty good understanding here. "Raw record positions"
seem to fit better. (This also leads well to a discussion why you cannot
rely on the record position to persist after a backup/restore or after your
transaction has committed). DB Keys are built from
 <relation number><page number><slot number>
 where page # is relative to relation, and slot # relative to page. This is
why the difference between DBKEY values is not uniform.
 So, the first lesson to learn is that db_keys don't have to be
monotonically increasing, that they are raw positions related to the
database itself and not to the operating system's addresses and that they
```

```
change after a backup and subsequent restore.
```

If you are interested in an analogy for db_keys, you should read about RIDs (Record Indentifiers) in SQL Server® and RowIDs in Oracle®. However, in SQL Server®, the record identifier is only available to database drivers and not to stored procedures, whereas in Oracle® it's exposed to general programmers as it's the case with InterBase®.

back to top of page

Probably the first question that comes to mind are why a person would want to use db_keys and when and how long they are valid in the database. In a general answer, people come to low levels when they can't do what they want in higher levels of abstractions or when they need better performance than in standard SQL. Now, on InterBase® specifically:

a. A db_key is faster than even a primary key (PK). Don't get surprised: a PK must be mapped through an index into a raw position, but a db_key is already this raw record position.

b. If for some special reason a table has no PK or the indexes are inactive (so exact duplicates may exist), db_keys are the only way to identify univocally each record in a table.

c. There are several statements that run faster when put in a stored procedure using a db_key than using the original SQL sentence. The typical examples are updates and deletions with complex conditions.

d. Some tricks can be made to write special statements that would require procedural logic instead of declarative logic (SQL) to be resolved without db_keys.

The duration of db_keys is a point that's not very clear in the manuals unless you read with sharp eye the API Reference Guide. By default, a db_key is valid only for the duration of the current transaction. After you use Commit or Rollback, the db_key values you had are inaccurate. If you are using auto-commit, you still can use db_keys because this mode uses an implicit CommitRetaining, that means the transaction context is retained, so garbage collection cannot proceed and hence, the db_keys used remain valid until you do a “hard” Commit.

At the risk of being repetitive, the explanation is when a transaction is committed, garbage collection may proceed so old record versions can be collected and “compacted”, namely, marked as free to use. Also, another transaction might have deleted a record you were using but your transaction isolation level prevented you from being aware of this change. Your db_key grabbed when the transaction was active can point now to a record version that doesn't exist anymore. If you took a lot of time to do your update to the record identified with the db_key, you'll want to check that the record has not being changed by another transaction in the meantime. InterBase® helps tracking some of these conflicts, see Writing behavior among conflicting transactions.

You can change the default duration of db_key values at connection time by using the API or IBObjects. You can specify that the values must be kept along the entire session; this means across the time elapsed between the time you connect to the database and the time you disconnect. However, keep in mind that this means garbage collection is stuck for all this time. Internally, InterBase® keeps a transaction open that stops garbage collection. In highly interactive environments, this may result in your database file growing at a big rate and the operations in the database being each time slower. As a corollary, having an auto-commit transaction working for all the day or using only CommitRetaining or simply opening a transaction that's not committed for several hours without need is not a wise solution.

Now, we will analyze briefly the four points on using transactions that were shown above as possible advantages or workarounds:

Point a) A db_key is faster than a PK. One level less of indirection. However, if the database pages needed are in the main memory and the subset requested exhibits locality (all the records are near to the others) and it's small, the difference in access speed shouldn't be noticeable.

Point b) Sometimes, indexes are turned off or not defined at all for efficiency reasons. For example, maybe a table used as an audit trail won't be indexed so it imposes a few penalties for inserting actions and data related to the operations on other tables. Should you need to make interactive alterations to the data, your only point of identification for a record is its raw position, i.e. the db_key. When there's no key, IBObjects automatically uses the db_key. This method works well with updates and deletions. However, newly inserted records cannot be shown: how do you know what's the value of the db_key assigned to the new record? You only can refresh your query and fetch again the records.

Point c) The InterBase® optimizer stills has trouble with some sentences. If you try to run something like

```
update tableA A
set sumfield = (select sum(B.valfield)
from tableB B where B.FK = A.PK)
where condition
```

against a huge table, you are likely to find performance problems. If you run the same operation often, then it's worth the time to write a stored procedure doing

```
for select B.FK, sum(B.valfield) from tableB B
group by B.FK
into :b_fk, :sum_vf do
   update tableA A set sumfield = :sum_vf
   where A.PK= :b_fk and condition
```

Although speedier, it's still has the problem that records in A have to be located by the PK each time a new pass of the FOR/DO loop happens. Some people claim better results with this alien syntax:

```
for select [B.FK,] sum(B.valfield), A.rdb$db_key
from tableB B join tableA A on A.PK=B.FK
where condition
group by B.FK
into [:b_fk,] :sum_vf, :dbk do
   update tableA A set sumfield = :sum_vf
   where A.rdb$db_key = :dbk
```

The last form seems to have some advantages:

- First, the filtering of the common records for A and B can be done in an efficient manner if the optimizer can make a good filter from the explicit JOIN.
- Second, the WHERE follows to make additional filtering (it may be appended by the user with and AND to the basic JOIN condition depending on the clause contained in the WHERE) before the update checks its own condition.

- Third, the dependent table (A) has its records located by raw db_key values, extracted at the time of the JOIN, so it's faster than the looking for through the PK. I put two parts between square brackets because InterBase® 6 doesn't require them but older InterBase® versions might insist that the field used in the GROUP BY must be present in the SELECT clause and the second optional part is dependant on the former because all fields described in the SELECT must be received in a variable after the INTO clause, in the order of appearance. As aforementioned, dbk, the variable holding the db_key value in the example, must be defined as char(8) because we are dealing with a table.

Usually, it's not possible to use an explicit query plan in insertions, updates or deletes, so the only way to proceed seems to be using a procedure in case you get slow performance. You may wonder why I include insertions. Well, an insert statement is not contrived to a single sequence of values that's sent by the client application. In fact, it can be very complicated as *insert into this table the summary, average and count of distinct values from the join of these two tables while condition C.e=somevalue is met but only if the item inserted is not already in this table, otherwise perform an update.* There's an special syntax for inserting the results of a SELECT statement:

```
insert into tableA
select C.pkey, sum(B.a), avg(B.b), count(distinct C.c)
from tableB B join tableC C on B.a = C.h and C.e = somevalue
where C.pkey not in (select pkey from tableA)
group by C.pkey
```

Of course, the update part has to be left for another sentence. You can change

```
where C.pkey not in (select pkey from A)
```

to be

```
where exists(select pkey from tableA A where A.pkey = C.pkey)
```

for a measure of relative performance. It's possible to rewrite the whole statement with a stored procedure. You have to test if this is faster or slower and remember that the statement shown above only covers the insert portion, not the update portion as in the following procedure:

```
for select C.pkey, sum(B.a), avg(B.b), count(distinct C.c),
from tableB B join tableC C on B.a = C.h and C.e = somevalue
group by C.pkey,
into :c_key, :sum_a, :avg_b, :count_c do
begin
   select A.rdb$db_key from tableA A where A.pkey = :c_key
   into :dbk;
   [if sum_a is NULL then sum_a = 0;
   if avg_b is NULL then avg_b = 0;]
   if (dbk is NULL)
   then insert into
   tableA(pkey, summary, average, count_of_distinct)
   values(:c_key, :sum_a, :avg_b, :count_c);
   else update tableA A set
   summary = summary + :sum_a, average = average + :avg_a,
   count_of_distinct = count_of_distinct + :count_c
```

```
    where A.rdb$db_key = :dbk;
end
```

It might be the case that B.a and B.b are allowed to be NULL and in this case, this procedure would be the only native way of dealing with that condition before the insert or update. This is the purpose of the two lines between square brackets to mean they must be enabled if NULLs can occur. Remember that the accepted comment delimiters in InterBase® procedures and triggers are the same than in language C, so if you really want to leave these lines in the code but without effect, you must enclose them between /* and */ to be bypassed. Conversely, COUNT() never can be NULL, because it only counts records. This is a rule you'll want to remember:

- When you make a COUNT(*) in a table, you are counting all records.
- When you make a COUNT(FIELD), you are counting all records where field is not NULL.
- When you make a COUNT(DISTINCT FIELD), you are counting only the different values in that field, namely, all repetitions of the same value account for one item. Warning: if the field allows NULLs, they are ignored regardless of how many NULLs are, so all repetitions of NULL don't account for one item.

If you don't mind writing convoluted sentences, you can count all NULL as one occurrence more in a “distinct”

```
clause with a weird syntax like this:
select count(distinct table.field) +
(select count(*) from rdb$database
where (select count(*) from table t where t.field is null)>0)
from table
```

or in a more standard way:

```
select count(distinct table.field) +
(select count(*) from rdb$database
where exists (select * from table t where t.field is null))
from table
```

So, I hope you understood that in the worst case, the result of count is zero, but it can't be NULL or negative.

back to top of page

One of the things that most confuses people not used to db_key is the fact almost all the time it needs to be qualified explicitly with the table name. For example,

```
select rdb$db_key from table
```

will work without problems, but

```
select rdb$db_key, field from table
```

won't work even if it's your birthday. In this case, you need explicit qualification:

```
select table.rdb$db_key, field from table
```

When the name of the table is long or convoluted or there are more than one table, table aliases come in handy to make sentences more clear as it was shown earlier. Table aliases are treated as a synonym of the table you're using, so you can put them wherever you'll put the original table's name. In this case “this is my table” is the real name whereas t is a synonym. The real name has to be surrounded by double quotes because it has spaces. This name is only possible in InterBase® 6 using dialect 3:

```
select t.rdb$db_key, t.pkey, t.oxygen from "this is my table" t
```

and the sample applies when you need db_key plus all other fields, with a little additional trick:

```
select t.rdb$db_key, t.* from "this is my table" t
```

Here, if you intend to put only an asterisk, you'll get a syntax error. Beware that the use of asterisk is discouraged, because the order of fields may change as the table is restructured or new and unexpected fields may appear that would render a FOR/DO/INTO construction useless in a stored procedure (of course, the engine has some protection against changing objects that have dependant objects as it's the case of a table with a procedure using it, but don't count on this protection as being 100% foolproof).

You should note that, as was stated earlier, db_keys are per table and use 8 bytes (except in views, as already mentioned), so if you want to use that property on all tables of a JOIN, you must read each db_key; for example:

```
for select c.id, c.rdb$db_key, s.id, s.rdb$db_key
from cats c join squirrels s
on c.owner = s.owner
into :cat_dbk, :squi_dbk do
begin
   update cats set partner_squirrel = s.id
   where cats.rdb$db_key = cat_dbk;
   update squirrels set partner_cat = c.id
   where squirrels.rdb$db_key = squi_dbk;
end
```

Note the example is very contrived and nonsense: there's no guarantee a person has to have only one squirrel and only one cat at a maximum, so the link has to be made really through a third table to support the ↔N relationship, namely, a NUB. (By the way, do you need to link cats and squirrels with the same owner?) However, using this construction you can improve the performance of operations that need to update two or more tables at the same time and usually with interdependent information. Had the procedure being written in a typical DSQL way, it would have required two separate SQL update statements with an inner select and furthermore, you cannot take easy control of a query plan that's the inner part of an insert, delete or update statement. Using a procedure instead, it can be written in a traditional form, using the retrieved primary keys (c.id and s.id) in the WHERE clause, but would be slower as the engine would have performed a search by PK or a natural scan, whereas with db_keys as shown above, the procedure runs as fast as allowed by the engine, because it locates each record by its position that was obtained in the select part.

I have stated in the prior paragraph that's when a query is nested in another SQL statement, you

cannot take easy control of the nested query's plan. One reason is InterBase® generates two separate plans for these sentences (one for the outer sentence and other for the inner sentence) and the syntax to force an explicit SQL plan doesn't cater for more than one plan. To be accurate, you can if you put each plan after the sentence where it's used, but probably the example doesn't look very attractive, using the example employee database that comes with InterBase®:

```
select emp_no, dept_no,
(select department from department d
where d.dept_no = e.dept_no
PLAN (DEPARTMENT INDEX (RDB$PRIMARY5))
) as dept_name
from employee e
PLAN (E NATURAL)
```

Furthermore, using a plan that a human being estimates to be faster can be simply rejected by the InterBase® optimizer because it has some “prejudices” about bad and good plans. As the outer an inner sentences become more complex, formulating by hand a better plan and having the optimizer to agree on it starts to be a really tricky exercise not recommended for the faint of heart.

Until now, examples of insertions and updates have been shown. Now, an example of a deletion is presented. The idea is very simple:

```
delete from employee
where emp_no not in (select emp_no from employee_project)
and job_code in ('Eng', 'Mngr')
```

So we are mimicking a company that wants to lower costs by dropping employees with no current project assignments. InterBase® will answer with these plans in the example database employee.gdb:

```
PLAN (EMPLOYEE_PROJECT INDEX (RDB$PRIMARY14))
PLAN (EMPLOYEE INDEX (RDB$FOREIGN9,RDB$FOREIGN9))
```

However, the task is not completed: we need to delete the salary history or these employees won't be deleted by InterBase® to maintain referential integrity. We would try to assign these data to a special employee, but in this case we would wipe out the history too and this must be done before the other statement:

```
delete from salary_history
where emp_no in
(select emp_no from employee
where emp_no not in (select emp_no from employee_project)
and job_code in ('Eng', 'Mngr'))
```

so we are converting the DELETE previously outlined now as a SELECT and using them to find what salary_history entries to delete. Why the nesting? Because job_code is an attribute of employee and not from employee_project, so if there's a shortcut, it's not obvious. InterBase® will answer with this plan:

```
PLAN (EMPLOYEE_PROJECT INDEX (RDB$PRIMARY14))
```

```
PLAN (EMPLOYEE INDEX (RDB$PRIMARY7,RDB$FOREIGN9,RDB$FOREIGN9))
PLAN (SALARY_HISTORY NATURAL)
```

Doesn't look a very simple SQL plan, right? Someone should try to optimize these two statements or deleting idle employees will cause all employees going idle while the server eats resources trying to delete all needed entries. The main part is the same in both sentences, but one uses DELETE and the other uses SELECT. We can make them only one in a stored procedure:

```
for select emp_no from employee
where emp_no not in (select emp_no from employee_project)
and job_code in ('Eng', 'Mngr')
into :emp_no do
begin
   delete from salary_history where emp_no = :emp_no;
   delete from employee where emp_no = :emp_no;
end;
```

Probably runs faster, but it's not optimized yet. First. the NOT IN test is slow. Who said it cannot be converted to a join? Let's see:

```
select e.emp_no from employee e
left join employee_project ep
on e.emp_no = ep.emp_no
where e.job_code in ('Eng', 'Mngr')
and ep.emp_no is NULL
```

This part looks promising because InterBase® is using all available indexes:

```
PLAN JOIN (E INDEX (RDB$FOREIGN9,RDB$FOREIGN9),EP INDEX (RDB$PRIMARY14))
```

If you change

```
where e.job code in ('Eng', 'Mngr')
```

to be instead

```
where e.job_code = 'Eng' or e.job_code = 'Mngr'
```

the plan changes to become

```
PLAN JOIN (E NATURAL,EP INDEX (RDB$PRIMARY14))
```

and this means if you prefer the OR to the IN construction, InterBase® does a natural scan (sequential scan) of the employees, looking for the ones that are either in engineering or management. The use of the index thanks to the IN clause can be a benefit when people in engineering and management are a few compared with the total amount of employees; otherwise, the natural scan seems to be better.

Note there's a catch: actually the test for 'Eng' and 'Mngr' became part of the WHERE and not a second condition AND'ed with the JOIN condition. If you use it as a second condition in the JOIN, you

get a completely different SQL plan and your results aren't the expected at first glance: the condition won't filter anything... you could be dropping busy employees in a real case. This is not a bug in InterBase® but the way outer joins work! The moral is: watch your plans and your results when doing these conversions if there is a left or right join playing in the game.

As the procedure is defined, it moved the problem inside the loop: now we are deleting each employee, seeking it by primary key, so let's apply the last step:

```
 create procedure del_idle_employees
 as declare variable dbk char(8);
 begin
    for select e.rdb$db_key, e.emp_no from employee e
    left join employee_project ep
    on e.emp_no = ep.emp_no
    where e.job_code in ('Eng', 'Mngr')
    and ep.emp_no is NULL
    into :dbk, :emp_no do
    begin
       delete from salary_history where emp_no = :emp_no;
       delete from employee where rdb$db_key = :dbk;
    end;
end
```

If you are still wondering what has been improved after these successive steps, then let's check the most important achievements:

Initially, we needed two loops and the one to clean salary_history was not simpler than the code put in the final procedure.

Nested queries are expensive in InterBase®. They are much better handled as joins. Even a NOT IN clause can be faster if rewritten as a LEFT JOIN as shown above. There are exceptions, of course.

Salary_history must use the primary key of employee to locate records because there can be several records per employee as his/her salary has changed. Employee can take advantage of rdb$db_key to delete the record found in the join no index overhead.

In case it's not clear enough, rdb$db_key cannot be used across tables: even if you do a join of two tables using indexed fields, you cannot compare rdb$db_key. In the example, above, there's a join between employee and employee_project. Even if it was an inner join (the default join type) instead of an outer join (left, right or full join), when e.emp_no=ep.emp_no is met, e.rdb$db_key<>ep.rdb$db_key will be true: rdb$db_key is a raw record position and therefore, two tables cannot use the same physical space in the database.

back to top of page

As in any practical case, the extra power of db_key is needed when the utmost performance is required. In the last example, if there are 42 employees as it's the case of the sample employee.gdb database that comes with InterBase®, using db_key to find and delete employees is clearly overkill. The idea was to use an example with a database that any InterBase® user can play with. But in real life, there are cases with multiple dependencies on tables that have millions of records and in this case, taking time to write a small stored procedure to automate and optimize a job that could be ran several times a month is an effort that pays for itself.

Point d) There are some interesting tricks. For example, let's assume we want to make a column unique, but there are several records that have the same value on this field (column). We will simply get rid of the duplicates and leave only one record with each different value. Let's assume the column is already defined as NOT NULL (if it was accepting nulls, the default at definition time, changing it to NOT NULL is another story). A typical table like this needs some amendment:

```
create table repet(a int not null);
```

However, trying to put a PK fails because there's repeated values. We can build a stored procedure to get rid of duplicates. But there's a neat trick from Ruslan Strelba, in his own words use the undocumented rdb$db_key column so you can use such a statement and this is the tip:

```
select
c0.some_value
from your_table c0, your_table c1
where
c0.some_value=c1.some_value and
c0.rdb$db_key>c1.rdb$db_key
```

This works when you have two occurrences of the same value in the table: it will show only one record, so you can wipe it out and you'll be left with one occurrence. However, keep in mind that this trick doesn't work when a value is repeated three or more times. You can put this statement in a stored procedure and it will clean the table:

```
for select
c0.rdb$db_key
from your_table c0 join your_table c1
on c0.some_value=c1.some_value and
c0.rdb$db_key>c1.rdb$db_key
into :dbk do
   delete from your_table yt where yt.rdb$db_key  = :dbk;
```

There's no need of a BEGIN ... END block in the loop because DELETE is the only instruction. Now, let's assume you don't want to build a stored procedure. First, let's use the original sentence but with the table repet shown previously:

```
delete from repet r
where r.rdb$db_key in (
select c0.rdb$db_key
from repet c0, repet c1
where
c0.a=c1.a
and c0.rdb$db_key>c1.rdb$db_key)
```

But after executing it, oh surprise, no record has been deleted! Whether this is a bug on the InterBase® optimizer or an esoteric problem is left as an exercise. I will take advantage of this to preach the use of the EXPLICIT JOIN SYNTAX, and we will see that this time it succeeds:

```
delete from repet r
where r.rdb$db_key in (
```

```
select c0.rdb$db_key
from repet c0 join repet c1
on c0.a=c1.a
and c0.rdb$db_key>c1.rdb$db_key)
```

Probably you are interested in getting a listing of records to delete leaving only one representative of each repeated value, no matter how many times it appears. I've found this is the general sentence that must be put in the procedure instead of the one shown above:

```
for select distinct c0.a, c0.rdb$db_key
from repet c0 join repet c1
on c0.a=c1.a and
c0.rdb$db_key>c1.rdb$db_key
into :dummy, :dbk do
   delete from your_table yt where yt.rdb$db_key  = :dbk;
```

The other alternative is to build a stored procedure that walks the table in order and always skips the first value of a group of identical values, but this requires a flag and a temporal variable.

Now on another example with data you can test for yourself. I've decided again to use a the table salary history, because it's primary key is

```
(EMP_NO, CHANGE_DATE, UPDATER_ID)
```

This table is intended to keep the historical salaries that an employee has had while working for the company. Of course, the employee's number doesn't suffice to make up a PK, because the same employee will appear several times as his/her salary changes. The identification of the user_id making the update is part of the PK for preserving consistency. Now, our historical table has grown too much so we backed up its information and intent to leave only the most recent change for each employee. Hence, the simpler way is to ensure that emp_no is not repeated. First, we want to know how many repetitions are:

```
select count(*)-count(distinct emp_no) from salary_history
```

If you haven't fiddled before with this database, you should get 16 as the answer, so we know we have to delete 16 records. This can be accomplished easily in a grid, but what if a real case returns 1500, for example? Clearly, a manual operation would only be chosen by a bureaucratic DB Admin seeking for ways to justify his full-time contract. At risk of being boring, we will walk several alternatives:

1. A convoluted statement can be the most immediate solution. For each group of records that have the same emp_no, we need to preserve the one that has the most recent date. In terms of date comparisons, this means the one that has the higher date. By default, indexes are ascending, so getting the minimum is fast but getting the maximum won't get any help from the index. Fortunately, the designers of this sample employee.gdb database already had defined the index we need:

```
CREATE DESCENDING INDEX CHANGEX ON SALARY_HISTORY (CHANGE_DATE)
```

Now, the problem is to write the SQL statement itself. Clearly, we need a DELETE with a nested SELECT clause. This doesn't suffice, however: we need the maximum but for the employee that's

being deleted, not the maximum date of all the table, so the inner statement depends upon the outer statement. This is called a correlated query, although the common cases are two SELECTs, not a DELETE plus a SELECT statement.

```
delete from salary_history es
where change_date < (select max(change_date) from salary_history es2
   where es2.emp_no=es.emp_no)
```

and InterBase® will answer with the following plan:

```
PLAN (ES2 ORDER CHANGEX)
PLAN (ES NATURAL)
```

This looks good and bad. First, MAX is using the descending index for the inner select, but not for the outer scan to perform the deletion. It could use it, because the LESS THAN condition would walk again the same index to find the first value less than the inner select and then continue walking the index to retrieve the rest of values. Instead InterBase® has decided to make a natural scan, too slow if the table has thousands of records. The problem is the correlation, namely, the fact the inner SELECT depends on the outer statement. To verify that InterBase® can use such descending index with LESS THAN, just prepare the statement

```
delete from salary_history es
where change_date < 'today'
```

and InterBase® will answer PLAN (ES INDEX (CHANGEX)) so there's place for enhancements. However, in this case, such index doesn't help our goal, so we will leave it. Let's build a procedure, then:

```
set term ^;
create procedure del_old_salary_history
as
declare variable dbk char(8);
declare variable dbk_prev char(8);
declare variable emp_no smallint;
declare variable emp_no_prev smallint;
begin
   emp_no_prev = NULL;
   FOR select sh.rdb$db_key, emp_no
   from salary_history sh
   order by emp_no, change_date
   INTO :dbk, :emp_no
   DO BEGIN
      IF (emp_no = emp_no_prev)
      then delete from salary_history where rdb$db_key = :dbk_prev;
      emp_no_prev = emp_no;
      dbk_prev = dbk;
   END
end ^
set term ;^
```

Let's explain what it does: first, we need to wipe out all past salaries for a person, namely, all salaries less the newest for each employee. So, we order by emp_no and change_date. Since there's an index due to the PK that spans emp_no, change_date and updater_id, ordering even for emp_no only will cause the engine to use the index and hence the correct order of the three fields. Since a PK generates an ascending index automatically and we're using such index, we need to pick the latest record for each employee. This makes the logic more complex. We trap the db_key and the emp_no of the current_record. If the employee number has changed, we have a new employee. This means the previous record is the latest from the previous employee and the one we should keep. Since we track the prior employee and the previous db_key in auxiliary variables, we know what the previous record was. Conversely, if the employee number didn't change, then we have the same employee and the prior record (not the current one) can be deleted, since it was not the latest for such employee. This is the reason we keep the previous db_key to delete by position the prior record. Now, on boundary cases:

- The first record: we can assume that an employee number is a positive one. However, to make the case generic, the auxiliary variable is assigned NULL. Since emp_no is part of the PK, it cannot be NULL, hence the first record won't be deleted by accident, since any value compared with NULL is unknown and this means FALSE in practice. Using a negative value for the auxiliary variable works in this case, but in other case, negative values in the PK might be allowed, hence NULL is safer.
- The last record: when inspecting the last employee, there may be one or more entries for salary_history. Being N the latest record and N-1 the prior, we can see: if N-1 and N are records for the same employee, N-1 is deleted in the last iteration. Otherwise, N-1 is preserved. Whether N is the only record for the latest employee or not, it won't be compared with anything, since there won't be another iteration, hence it will be preserved.

It has been said that ordering by an index is not a very good idea, since the engine should read random pages and load them in the cache to satisfy each record in the order mandated by the index. However, the assumption in this case was different: each time our condition matches, we delete the record the was loaded previously. So, there's a great chance that such page is still loaded in memory when the deletion happens. Since we are using db_key, there's no index search. Also, since the procedure does all the job, the operation should be fast, so there's no much chance that other requests just unload our previously accessed page before we need it.