Version 3

- A lot has changed since NAV 2013. Now we don't have cursors anymore to slow our queries down, but we have MARS now.
  The type of cursor used made it almost forced SQL Server to use the index that is specified in the ORDER BY clause which is defined by the SETCURRENTKEY used in NAV. That means that if you specified a key because you needed the records in that order but your filters were on other fields, SQL Server was very inefficient to find the records you requested. And if the index did not exist (key property MaintainSQLIndex = FALSE), it generally scanned the whole table. Starting with NAV2013, the SETCURRENTKEY was just used for the ORDER BY clause but let SQL Server decide which index to use. With NAV2013R2, other possibilities where added like CALCSUMS on any decimal field with any filtered fields were possible without having a SIFT in NAV.

- I saw a lot of times that a lot of people use the different ways to get the data from the database in a not really performant way. Or they write statements that are not needed and this way they create some confusion in the code.
  This stimulated me to write this how-to to explain with examples how (not) to use the different statements and when (not) to use them.
  Most points are with SQL in mind. But some of them are also valid for native DB (you really should stop using the native DB!).

- PRE-NAV2013: means all versions before NAV2013. Including NAV2009xx RTC
- NAV2013+: means all versions starting with NAV2013
- I have tried to put all possibilities in the best order of performance. Meaning: if you need to do something (reading data, modifying data, deleting data,…), read this How-to from the beginning and for each bullet point, check if you can use it. If you can, it is probably the most efficient one for your case. If it is not useful, check the next bullet point.
- I have done my tests with NAV2009R2 classic and NAV2017 CU01 and some tests also in NAV2013,NAV2013R2,NAV2015,NAV2016.

- SELECTLATESTVERSION : In general NAV tries to read from its buffers (even the classic client has them, but I don't know how they behave). With this command, you force NAV to read data from SQL Server.

- CLEAR, INIT and RESET: Let's start with the difference between these 3:
  - TheTable.RESET: this is used to remove all FILTERS (also in different FILTERGROUP's) from a record-variable. It also resets the SETCURRENTKEY to the primary key.
    IMPORTANT: DO NOT USE this statement for clearing the fields in a record. It will NOT clear them.
    - VERY WRONG: If the record has some data in it before the RESET-statement, it will keep these values.
      ```
      TheTable.RESET;
      TheTable."Primary Key" := Some Value;
      TheTable."Field N" := Some Value;
      TheTable.INSERT(FALSE);
      ```
    - CORRECT:
      ```
      CLEAR(TheTable); // also clears the primary-keyfields
      TheTable."Primary Key" := Some Value;
      TheTable."Field N" := Some Value;
      TheTable.INSERT(FALSE);
      ```
  - TheTable.INIT: this statement clears all fields in the table EXCEPT THE PRIMARY-KEY FIELDS (and the timestamp field)! It does NOT touch the FILTERS or CURRENTKEY! So if

  you want to insert a new record, keeping the primary key-fields already in the record-variable, this is the command to use.
  - CLEAR(TheTable): this statement does a TheTable.RESET AND a TheTable.INIT AND clears the primary keyfields. I almost always use this statement to INSERT a new record. It is the only command that also resets the CHANGECOMPANY-statement. Also global variables defined inside the table will be cleared. BUT global temptables defined inside the table will NOT be cleared. To clear them, you need to create a method in the table object to clear them. (CLEAR also works on other object-types and the same is valid for global variables and temptables defined inside the object.)

- GET: This command is the best to use if you want to search the record using its primary key. You can also filter on the primary key and use FINDFIRST. The SELECT-statement sent to SQL is almost the same (the FINDFIRST adds a TOP 1 to the SELECT). But the GET requires less coding and is easier to read. No RESET-SETCURRENTKEY-SETRANGE are needed. The GET does NOT EVEN consider them.

  - WRONG (because it creates confusion and the GET does not consider the previous commands):
    ```
    TheTable.RESET;
    TheTable.SETCURRENTKEY(...)
    TheTable.SETRANGE/SETFILTER
    TheTable.GET(...);
    ```
  - CORRECT:
    ```
    TheTable.GET(...);
    ```

  - In case you want to do a GET and even if you don't have a record, you want to continue with your code, you can do this:
    - Version 1
      ```
      CLEAR(TheTable);
      IF TheTable.GET(....) THEN ;
      ```
    - Version 2
      ```
      IF NOT TheTable.GET(....) THEN
        CLEAR(TheTable);
      ```

    Both versions will have a clean record in case they don't find a record. No dirty date will be in it after this code. Personally I prefer the first version: I prefer FIRST to clean the record and THEN to get the data (if available).

- The other commands need filtering, so before I start explaining them, I give an good programming practice that is valid for all of them. To make the code easy to read/interpret/review/fix/... , it is best to put the commands influencing the filtering together starting with a RESET. This makes sure you never have to start searching if some other part of the object has put some filters on it and if these filters should be there or not in your part of the code. I always use them together. This makes sure to other programmers I didn't forget to put the SETCURRENTKEY.
  ```
  TheTable.RESET;
  TheTable.SETCURRENTKEY(...); // even if you want to use the primary key
  TheTable.SETRANGE(...)  or TheTable.SETFILTER(....)
  ```
  - NAV2013+ : use the SETCURRENTKEY only when you need to order the data in a certain order. Otherwise do not use it anymore.

- NAV2013+ service has a buffer for records shared by ALL sessions on the service. PRE-NAV2013: each session has its own buffer for records and to make things even worse: it is only 32-bit so its memory is limited to about 4GB. With the 64 bit of NAV2013+ you can even put the whole database in cache memory if you have enough memory in your server (and DO separate SQL Server from the NAV server and DON'T put terminal server on the NAV server either).

- The different other commands and what they do and how to use them
    - The following commands are examples to read data when you DON'T want to lock records.
    - ISEMPTY: This is THE statement to use if you want to check if there is at least one record in the (filtered) table but you don't need the record itself. It doesn’t return the record itself. It just returns a true or false.
      Examples:

      ```
      IF SomeRecord.ISEMPTY THEN BEGIN
              // I haven't found a record, so I need to do something
      END;

      IF NOT SomeRecord.ISEMPTY THEN BEGIN
              // I have found a record, so I need to do something but I don't need the record itself
      END;
      ```

    - COUNTAPPROX: This statement is used if you want an approximate count of the number of records in the (UNFILTERED) table. It can be used for a progressbar. It is lighter for the server to execute than the COUNT. Once you have a filter on it, it behaves like the COUNT.
        - PRE-NAV2013: I haven't tested it but NAV2009 help states it uses statistics on the table to get the info, so it is not necessary accurate.
        - NAV2013+: I checked only NAV2017 and with or without filtering, it behaves exactly like COUNT.
    - COUNT: This statement is used if you want a precise count of the number of records in the (filtered) table. Do NOT use it if you just want to know if there are no records or at least one record. In these cases, use ISEMPTY. The COUNT reads all records in the table (or if you are lucky, all entries in an index), so if there are a lot of records, this can be SSSSSLLLLLOOOOOWWWWW.
    - FINDFIRST: This is THE statement to use if you only need the first record. But NEVER use it if you need to loop records. It only returns 1 record even if there are more records in the filter. It issues a SELECT TOP 1 …
    - FINDLAST: This is THE statement to use if you only need the last record. But NEVER use it if you need to loop records. It only returns 1 record even if there are more records in the filter. It issues a SELECT TOP 1 … ORDER BY xxx DESC
    - FINDSET: I noticed a big difference between PRE-NAV2013 and NAV2013+!!!
        - PRE-NAV2013:
            - The SELECT has OPTION (OPTIMIZE FOR UNKNOWN) (Only available for SQL Server 2008 and later)
            - SELECT TOP N = defined by File=>Database=>Alter=>Tab Advanced=>Caching: Record set.

        - It does NOT create a cursor which is good. It starts with a SELECT TOP N. (Actually it is N + 1 but that is a detail. N or N+1 does not change the fundamentals)
        - If you are doing NEXT more than N times, at a certain point NAV does not have records in the buffer anymore and uses a CURSOR to read all the records. This SELECT does NOT have a TOP N and has it a filter to exclude the already read records.
        - Use this command if you know that the number of records read is generally smaller than N records or you generally read less than N records before exiting the loop.
        - You CANNOT use this command if you want ordering ASCENDING(FALSE)! Reason is the CURSOR that is used. If you try it, you will get a runtime error. In this case, use FIND('+') and NEXT(-1) or ASCENDING(FALSE) and FIND('-') and NEXT.
    - NAV2013+:
        - NO CURSOR!
        - The SELECT has OPTION(OPTIMIZE FOR UNKNOWN, FAST 50). The "50" is hardcoded so we do not have control over it (Well, I haven't found a way). The FAST is just telling SQL to find the first 50 records as fast as possible and send them over then find the rest of the records.
        - The SELECT does NOT contain a TOP N. So it returns ALL records in the filter.
        - Use this command if you know that you need all records in the filter.
        - You CAN use this command if you want ordering ASCENDING(FALSE).
    - So for small recordsets (less than 50 records), PRE-NAV2013 and NAV2013+ behave in the same way. If there are more records in the recordset, PRE-NAV2013 will first send a SELECT to SQL Server and then a CURSOR.

- FIND('-'): I noticed a big difference between PRE-NAV2013 and NAV2013+!!!
    - PRE-NAV2013:
        - A cursor is immediately created to read ALL records. The SELECT has OPTION (OPTIMIZE FOR UNKNOWN) (Only available for SQL Server 2008 and later)
        - SELECT TOP N = defined by File=>Database=>Alter=>Tab Advanced=>Caching: Record set.
        - Use this command if you know that the number of records read is generally larger than N records AND you generally read more than N records. The FINDSET would perform worse in this case because it sends first a SELECT TOP N and then it creates the cursor. The FIND('-') immediately creates the cursor.
        - You CAN use this command if you want ordering ASCENDING(FALSE).
    - NAV2013+:
        - NO CURSOR!
        - The SELECT has OPTION(OPTIMIZE FOR UNKNOWN, FAST 50). The "50" is hardcoded so we do not have control over it (Well, I haven't found a way). The FAST is just telling SQL to find the first 50 records as fast as possible and send them over then find the rest of the records.
        - It starts with a SELECT TOP 50.
        - If you are doing NEXT more than N times, at a certain point NAV does not have records in the buffer anymore and does a new SELECT that does NOT have a TOP N and has it a filter to exclude the already read records. So it behaves like the FINDSET of PRE-NAV2013 (except for the cursor-part).

    - Use this command ONLY if you know that the number of records is more than 50 records AND you generally read less than 50 records before exiting the loop.
    - You CAN use this command if you want ordering ASCENDING(FALSE).
- FIND('+'): like FIND('-') and NEXT(-1)

- FIND('='): This statement is a little strange. It almost works like the GET using the primary key values to find a record. You use it by filling up the fields of the primary key and only then doing a FIND('='). It ALSO considers the filters on the record! It can be useful if you already have the record you need in the variable, and you want to refresh it.

- FIND('>') and FIND('<'): This bases itself on the current values in the primary key of the record to get the next or previous record, it ALSO considering the filters. Better use NEXT or NEXT(-1).

- FIND('=<>'): This is the worst of all. It first tries FIND('=') and if it doesn't find a record, it does a FIND('<') and if it doesn't find anything, it does a FIND('>'). TO BE AVOIDED!

- Query object
  - NAV2013+ only
  - Avoids the loopy-loopy anti-pattern (REPEAT-UNTIL inside another REPEAT-UNTIL inside … [same for looping inside looping in reports and XMLports]). It is SSSSSSSLLLLLLLLOOOOOOOOWWWWWWW!!!!!!!
  - Query object data that is read is never cached, it is always read from SQL Server because a SQL Query is created and send to SQL Server to get the records in the most performance way. SELECTLATESTVERSION has no effect. Data can be cached in SQL Server.
  - Possibility to do create some specific keys to help with this SQL query.
    - You can even create a covering index for your table in the query. A covering index contains all fields in the anywhere in the complete SELECT statement. This means that if SQL Server uses the index it does not need to go to the real table to get some extra fields. Remember that when you create a secondary key in NAV, it always contains the primary key fields even if you didn't specify them.
    - NAV is even so intelligent (didn't check for NAV2013, but NAV2013R2 does it) that if you have a key with a SIFT in it, NAV will use the indexed view for that SIFT in its SQL statement
      - e.g. you need a SUM(Amount) per "G/L Account", "Document No." Create a key with "G/L Account", "Document No." with SIFT field Amount. Put property MaintainSQLIndex to FALSE if you don't need that index in SQL but keep MaintainSIFTIndex to TRUE. When your query is using this, NAV will send a query on the Indexed view and not on the G/L Entry Table. This will boost the performance of your query object even more.

EXAMPLES for reading data WITHOUT locking the records

Let it be said once and for all (I don't want to repeat it every time I use a REPEAT-UNTIL): This is REALLY BAD IN ANY CIRCUMSTANCE: NEVER use FINDFIRST/FINDLAST to start a REPEAT - UNTIL.

So if you see this

```
IF TheTable.FINDFIRST THEN // or FINDLAST + NEXT(-1)
  REPEAT
  UNTIL TheTable.NEXT = 0;
```

you know what to do! Change it!

- You need to check if there are ANY records in your (filtered) table, but you don't need any of them to work on. There are a lot of possibilities that all work, but performance can be a big problem.
  - (VERY) WRONG: because it counts ALL records
    ```
    IF TheTable.COUNT > 0 THEN ...  // or COUNTAPPROX
    ```
  - WRONG: It returns a SET of records to the client (TOP n on PRE-NAV2013 or all on NAV2013+)
    ```
    IF TheTable.FINDSET THEN ...
    ```
  - ALSO WRONG: Until some versions ago (I don't remember which), you would use this because there was no alternative. This opens a cursor on PRE-NAV2013 and returns a TOP 50 of records in NAV2013+.
    ```
    IF TheTable.FIND('-') THEN ...  // or FIND('+')
    ```
  - STILL WRONG: It doesn't open a cursor in SQL, but it still returns 1 record (if there is at least 1 available)
    ```
    IF TheTable.FINDFIRST THEN ... // or FINDLAST
    ```
  - CORRECT: it NEVER returns a record. It just returns a boolean to say there are/are not records.
    ```
    IF NOT TheTable.ISEMPTY THEN ...
    ```
- You want ONY the first/last record (if it exists), but NEVER more.
  - VERY WRONG: It returns a set of records to the client (TOP n on PRE-NAV2013 or ALL records on NAV2013+)
    ```
    If TheTable.FINDSET THEN ...  // to get the last record , you would
    need to put ASCENDING(FALSE) before calling FINDSET (later more on
    this because it is even MORE WRONG than VERY WRONG)
    ```
  - ALSO VERY WRONG: This opens a cursor on PRE-NAV2013 or a TOP 50 set on NAV2013+.
    ```
    If TheTable.FIND('-') THEN ...  // or FIND('+')
    ```
  - CORRECT: it doesn't open a cursor and it returns only 1 record
    ```
    If TheTable.FINDFIRST THEN ... // or FINDLAST
    ```
- You want ALL records in ASCENDING order
  - PRE-NAV2013
    - You are quite sure that almost always you will have less than N records (Yep. That parameter again!)
      ```
      IF TheTable.FINDSET THEN
        REPEAT
          ...
        UNTIL TheTable.NEXT = 0;
      ```
    - You are quite sure that almost always you will have more than N records (Yep. That parameter again!)
      ```
      IF TheTable.FIND('-') THEN
      ```

```
  REPEAT
    ...
  UNTIL TheTable.NEXT = 0;
```

  - NAV2013+
    - Whatever the number of records

      ```
      IF TheTable.FINDSET THEN
        REPEAT
          ...
        UNTIL TheTable.NEXT = 0;
      ```

- You want ALL records in DESCENDING order
  - PRE-NAV2013
    - You are quite sure that almost always you will have less than N records (Yep. That parameter again!)

      ```
      ASCENDING(FALSE);
      IF TheTable.FIND('-') THEN
        REPEAT
          ...
        UNTIL TheTable.NEXT = 0;
      ```

      ```
      FINDSET and ASCENDING(FALSE) is not possible and would error out!
      ```

    - You are quite sure that almost always you will have more than N records (Yep. That parameter again!)

      ```
      ASCENDING(FALSE);
      IF TheTable.FIND('-') THEN
        REPEAT
          ...
        UNTIL TheTable.NEXT = 0;
      ```

  - NAV2013+
    - Whatever the number of records

      ```
      ASCENDING(FALSE);
      IF TheTable.FINDSET THEN
        REPEAT
          ...
        UNTIL TheTable.NEXT = 0;
      ```

- You want SOME but not all records in ASCENDING order
  - PRE-NAV2013
    - You are quite sure that almost always you will read less than N records (Yep. That parameter again!)

      ```
      IF TheTable.FINDSET THEN
        REPEAT
          ...
        UNTIL TheTable.NEXT = 0;
      ```

    - You are quite sure that almost always you will read more than N records (Yep. That parameter again!)

      ```
      IF TheTable.FIND('-') THEN
        REPEAT
          ...
        UNTIL TheTable.NEXT = 0;
      ```

  - NAV2013+
    - You are quite sure that almost always you will read less than 50 records

      ```
      IF TheTable.FIND('-') THEN
        REPEAT
          ...
        UNTIL TheTable.NEXT = 0;
      ```

```
The FIND('-') issues a SELECT TOP 50
```

    - You are quite sure that almost always you will read more than 50 records

```
IF TheTable.FINDSET THEN
  REPEAT
    ...
  UNTIL TheTable.NEXT = 0;

With FIND('-') you will only read the first 50 records and then NAV
will issue another SELECT to read all records (filtering out the
records already read)
```

  - As you can see, between PRE_NAV2013 and NAV2013+ the way to read records in this way is inverted!

- You read records and need also the value of 1 or more flowfields.
  - PRE-NAV2013
    - No other way. For each record that is read, you do a CALCFIELDS and with that send another request to SQL Server.

```
IF TheTable.FINDSET THEN
  REPEAT
    TheTable.CALCFIELDS("Some Field");
    …
  UNTIL TheTable.NEXT = 0;
```

  - NAV2013+
    - Finally, we have smartqueries! NAV sends only 1 command to SQL which also reads the data for the flowfield.

```
TheTable.SETAUTOCALCFIELDS("Some Field");
IF TheTable.FINDSET THEN
  REPEAT
     …
     UNTIL TheTable.NEXT = 0;
```

  - Read the base table
    - PRE-NAV2013
      - You need a SIFT defined on it. It is possible NOT to maintain the SIFT in SQL (key property MaintainSIFTIndex=No). Of course, the real table will be read, so it could be quite slow if you don't maintain the SIFT.

```
TheBaseTable.CALCSUMS("the Field");
```

    - NAV2013+
      - You do NOT need a SIFT defined on it. But in that case the real table will be read…

```
TheBaseTable.CALCSUMS("the Field");
```

  - IMPORTANT to keep in mind. NAV will use the first SIFT it finds that has all the fields it needs.
    - Eg. You have a table with 2 SIFT on it:
      - "Customer No.","Item No." with SIFT : "Amount"
      - "Item No.","Customer No." with SIFT : "Amount"
      - You put a filter on "Item No." to calculate "Amount". NAV will use the first Key and will scan the whole SIFT-table (PRE-NAV2013) or indexed view (NAV2013+). The result will be correct, but it will be slow. It would be ok if you also (or only) have a filter on "Customer No.".

- You want the total inventory of a certain item in a certain location. There are 2 methods to do the same thing. Both have pro- and contra.
  - CORRECT:

        - PRO : you don't need to know the key
        - CONTRA: normal fields you have to assign a value (or do a GET) and flowfilters you need to filter

          ```
          CLEAR(Item);
          Item."No." := '1000';  // or also Item.GET('1000');
          Item.SETRANGE("Location Filter",'BLU');
          Item.CALCFIELDS(Inventory);
          MESSAGE('%1',Item.Inventory);
          ```
    - CORRECT:
        - PRO : Always filtering and not assigning values
        - PRE-NAV2013
            - CONTRA : You need to know the key on which the SIFT-field has been defined otherwise you get an error.
        - NAV2013+
            - PRO : You DON'T need a key! But it might be slower.

          ```
          ItemLedgerEntry.RESET;
          ItemLedgerEntry.SETCURRENTKEY("Item no.","Location Code");
          ItemLedgerEntry.SETRANGE("Item No."','1000');
          ItemLedgerEntry.SETRANGE("Location Code",'BLU');
          ItemLedgerEntry.CALCSUMS(Inventory);
          MESSAGE('%1',ItemLedgerEntry.Quantity);
          ```

- Using filtergroups: Filtergroups are useful for putting multiple filters on the same field or to hide filters from users.
    - Some advice: Only start using filtergroups from 10 and up. Filtergroups 0 to 6 are reserved (see the C/SIDE reference guide for more info), but I would keep some extra filtergroups free for the future (until some [a lot by now] versions ago, filtergroup 6 was NOT reserved).
    - Important: multiple filters on the same field behave as AND between them.
    - Example 1: you want to hide a filter from the user to avoid he can change it

      ```
      TheTable.RESET;
      TheTable.SETCURRENTKEY(....);
      TheTable.FILTERGROUP(10); // changes to a filtergroup that the user
      will NOT see
      TheTable.SETRANGE(....);
      TheTable.FILTERGROUP(0); // change back to the default filtergroup.
      The one in which the user may do want he wants
      FORM.RUNMODAL(0,TheTable); // the user will not be able to change the
      filters
      ```

    - Example 2: You need to put multiple filters on a field. E.g. You want all records in which a certain field starts with an A and ends with a B and somewhere in the middle is a C.
        - Of course you might create this SETFILTER. But how would you SETFILTER more complex queries?

          ```
          TheTable.RESET;
          TheTable.SETCURRENTKEY(...);
          TheTable.SETFILTER("The Field",'A*C*B');
          IF TheTable.FINDSET THEN
          ```

        - The same example with FILTERGROUP

          ```
          TheTable.RESET;
          TheTable.SETCURRENTKEY(...);
          TheTable.FILTERGROUP(10);
          TheTable.SETFILTER("The Field",'A*');
          TheTable.FILTERGROUP(11);
          TheTable.SETFILTER("The Field",'*C*');
          TheTable.FILTERGROUP(12);
          TheTable.SETFILTER("The Field",'*B');
          TheTable.FILTERGROUP(0);
          IF TheTable.FINDSET THEN
          ```

  - FILTERGROUP(-1): This only works from NAV2015 and up. It lets you use an OR statement while filtering on different fields.
    - This filters customer-records where the Name has a z in it OR Contact has a z in it or has EUR as the currency code. I put this code in the OnOpenPage-trigger. The messages came up as expected but when the page was opened, it showed ALL records. So it seems that you can only use it programmatically to loop records but NOT to show records.

      ```
      RESET;
      FILTERGROUP(-1);
      SETFILTER(Name,'@*z*');
      SETFILTER(Contact,'@*z*');
      SETRANGE("Currency Code",'EUR');
      IF FINDSET THEN
        REPEAT

      MESSAGE('Name=%1\Contact=%2\Curr=%3',Name,Contact,"Currency Code");
        UNTIL NEXT = 0;
      ```
    - I haven't been able to create filters to show records with Name OR Contact with a z in it AND have EUR as currency code. So this doesn’t seem to work

      ```
      RESET;
      SETRANGE("Currency Code",'EUR');
      FILTERGROUP(-1);
      SETFILTER(Name,'@*z*');
      SETFILTER(Contact,'@*z*');
      IF FINDSET THEN
        REPEAT

      MESSAGE('Name=%1\Contact=%2\Curr=%3',Name,Contact,"Currency Code");
        UNTIL NEXT = 0;
      ```

Modifying Data (Insert/Modify/Delete)

- First and for all some general advice for writing to the DB
  - Start a transaction as late as possible (meaning : first read all the data without locking anything and after that, start writing).
  - Lock as little as possible (if you change 1 record in a 10M records table, why lock all those records if you can only lock 1?).
  - Make your transactions as fast as possible (putting SLEEP-statements in a transaction is VERY (!) BAD. A CONFIRM or STRMENU is probably even worse…).
  - Make your transactions as small as possible (ok, not too small : if you need to change 1000 records in a table, don't put a COMMIT after each record).
  - Make your transaction so, that each transaction contains all writes it should do in 1 block or not at all.
  - Writing to a temptable does NOT start a transaction. And when an ERROR is raised, the writes to a temptable are NOT undone. You are in complete control of the temptable!
- IMPORTANT REMARKS:
  - LOCKTABLE (or FINDSET(TRUE,...)) works differently from LOCKTABLE in the native DB. In the native DB it blocks the table completely. In SQL, it means that from that point on, all

records read in that table will be exclusively locked, BUT it will not lock records in other tables. It is also possible some other records are locked because of the locking mechanism in SQL on which we have no control at all (at least not in C/AL). SQL can decide escalate from record locks to page locks and even to locking the whole table all depending on the number of records you lock. There are rules for that, but from a developers view: you don't know when SQL will do it.

  - NAV will, as a default, send SELECT … READUNCOMMITTED statements to SQL Server. This means that it will also do dirty reads (= it will also read data that has been changed by another session but NOT YET COMMITTED TO THE DATABASE. This means that after you read the record, it may be ROLLBACKed by the other session!)
  - READUNCOMMITTED = NO-LOCK (if you write TSQL statements, use READUNCOMMITTED because it really says what it does: it also reads uncommitted data. NO-LOCK sounds like you read data without locking it but it doesn’t tell you that it also reads uncommitted data. A lot of people fall in this trap!)
  - An explicit COMMIT releases all the locks you have on the records and commits the data to the database so someone else can lock the data.
  - An implicit COMMIT happens when the all the code you are currently running is finished. It does NOT happen at the end of the function you are running or when the code leaves the current object. It is only done when the GUI frees up again (CONFIRM and STRMENU do not free up the GUI! So never use them inside a transaction!) or your webservice call/job/background session stops processing C/AL code.
  - When you try to read a record locking it, you will wait until no one is locking the record anymore or until a certain amount (default=10 seconds ; setup : File=Database=>Alter=>Tab Advanced=>"Timeout Duration (sec)") of time has passed. In the latter case, you will get an error saying that the record is locked by someone else.

- You want to modify 1 record
  - PRE-NAV2013
    - WRONG : This will generate 2 (!) SELECT statements in SQL. The first SELECT is a READUNCOMMITTED and will be executed for the GET-statement. The second SELECT will be EXCLUSIVE-LOCK and will be executed for the MODIFY. After this an UPDATE-statement will be executed. The second SELECT is needed to exclusively lock the record, to check if no one else changed the record between the GET statement and the MODIFY statement.

      ```
      // You don't have a LOCKTABLE on it (a LOCKTABLE on another
      table will NOT lock current table
      TheTable.GET(...);  // or also FINDFIRST,FINDLAST
      TheTable."Some Field" := 'Some Value';
      TheTable.MODIFY(FALSE);
      ```
    - CORRECT: The GET will generate a SELECT with EXCLUSIVE-LOCK. The MODIFY will generate an UPDATE-statement

      ```
      TheTable.LOCKTABLE;
      TheTable.GET(...);  // or also FINDFIRST,FINDLAST
      TheTable."Some Field" := 'Some Value';
      TheTable.MODIFY(FALSE);
      ```
  - NAV2013+ (I didn't check it with NAV2013, but NAV2013R2 and NAV2017 have this behaviour, so I suppose also NAV2013 has it)
    - In NAV2013+, both statements (the ones from PRE-NAV2013) are CORRECT depending on your needs. So in certain cases you might want to use 1 or the other statement.

    - First a SELECT with READUNCOMMITED is send to the database. So it does NOT lock the record and it may also read an uncommitted version of the data. If you change it, the MODIFY will error out telling you someone else has changed the record. This is perfectly normal behaviour for NAV. Native DB behaved less-or-more like this. But not necessarily what you want. The MODIFY will send an UPDATE statement to SQL Server with a WHERE on the primary key fields and ALSO on the timestamp. So if it doesn’t find any record, it knows someone else has changed the record between the GET and the MODIFY.

When to use it? This is the best use when you probably will NOT change the record. So you don’t lock it unless you are really changing it. (If you do this in a loop, after the first record you change, the next records you read in this table will be locked.)

```
// You don't have a LOCKTABLE on it (a LOCKTABLE on another
table will NOT lock current table
TheTable.GET(...);  // or also FINDFIRST,FINDLAST
TheTable."Some Field" := 'Some Value';
TheTable.MODIFY(FALSE);
```

  - The GET will generate a SELECT with EXCLUSIVE-LOCK. The MODIFY will generate an UPDATE-statement. So you are sure you do NOT have dirty data and you lock the record so no one can change it until you release it (using a COMMIT or at the end of the code you are running).

    When to use it? When you are sure you will change the record and you know you haven't locked it yet.

```
TheTable.LOCKTABLE;
TheTable.GET(...);  // or also FINDFIRST,FINDLAST
TheTable."Some Field" := 'Some Value';
TheTable.MODIFY(FALSE);
```

- Using MODIFYALL to change 1 field in 1 or more records
  If you don't have the record, and you want to change just 1 field and you don’t need the record and you don’t care about the old value, there is a more performant way.
  This just sends an UPDATE statement to SQL Server and doesn’t need to fetch the record first from SQL Server and then send it back.
  You can also do this with filters that are not the primary key. But it will change the value for all records in the filters.
  Another interesting thing is that even if there are no records in the filter, no error will be generated, but a transaction will be started.
  If you need to change multiple fields in multiple records, it MIGHT be faster to do a MODIFYALL statement for each field independently. But this depends on a lot of variables (how many fields, how many records in the filter, the size of ther records, existing indexes that have those fields or indexes that that can be used to find the records, the weather [well, not really…], …).
  Of course, you can’t use this if the field is defined by other values in the table. It is 1 single value for all the records.

```
TheTable.RESET;
TheTable.SETRANGE(...)  // put a SETRANGE on all primary key fields
TheTable.MODIFYALL("Some Field",'Some Value',FALSE);
```

- You read a set of records and you need to change (almost) ALL records
  - First: be careful the order in which you read the records and which fields you change. The best is to leave the primary key order for reading records and then changing the fields. If you use a SETCURRENTKEY and you change a field in it, it is possible to skip records or read records multiple times. You can even have an endless loop. To my surprise, having FINDSET(TRUE,TRUE) (or LOCKTABLE + FIND('-')) or using a second variable does NOT help in either NAV-version! Remember : NO SETCURRENTKEY! If for some reason you need a SETCURRENTKEY and are changing one of its fields, go to the next bullet point!

  - Second: remember what I said about FINDFIRST/FINDLAST and REPEAT-UNTIL? No? Start reading from the beginning!
  - PRE-NAV2013
    - You are quite sure that almost always you will read less than N records (Yep. That parameter again!)
      You lock the table WITH the first TRUE in the FINDSET. The second TRUE is only needed in certain circumstances but I advise to use it always even if I didn't find any differences in the SQL statements sent by NAV. Probably some internal processing is different.

```
IF TheTable.FINDSET(TRUE,TRUE) THEN
  REPEAT
    TheTable."Some Field" := 'Some Value';
    TheTable.MODIFY(FALSE);
  UNTIL TheTable.NEXT = 0;
```

    - You are quite sure that almost always you will read more than N records (Yep. That parameter again!)

```
TheTable.LOCKTABLE;
IF TheTable.FIND('-') THEN
  REPEAT
    TheTable."Some Field" := 'Some Value';
    TheTable.MODIFY(FALSE);
  UNTIL TheTable.NEXT = 0;
```

  - NAV2013+
    - Using a second variable creates an extra SELECT to get the record in NAV2013+! Here we need to play with the FINDSET(TRUE,[TRUE]). So in this case, the best way is to use only 1 table-variable. And the first TRUE must be used. Best also put the second TRUE. As before I haven't found any difference in SQL statements sent by NAV.

```
IF TheTable.FINDSET(TRUE,TRUE) THEN
  REPEAT
    TheTable."Some Field" := 'Some Value';
    TheTable.MODIFY(FALSE);
  UNTIL TheTable.NEXT = 0;
```

- You read a set of records and you need to change SOME records (or previous bullet point was NOT usable)
  - Remember, in general you want to lock as little as possible and as late as possible! So the best way is to read all the records we need to check in the most performant way WITHOUT LOCKING them. The records we need to change we store in a temptable. Then we read the temptable and start modifying the records.

```
// You do NOT have a LOCKTABLE on table "TheTable"!
IF TheTable.FINDxxx THEN // See "EXAMPLES for reading data WITHOUT locking the records"
  REPEAT
      // save the records in a temptable and change it, but DON'T USE VALIDATE
      IF (Record has to be changed) THEN BEGIN
        TheTableTMP := TheTable;
        TheTableTMP."field N" := 'Some Value';
        TheTableTMP.INSERT(FALSE);
      END;
  UNTIL TheTable.NEXT = 0;

// now do a LOCKTABLE. This means that all records of that table that are read now, will be EXCLUSIVELY locked
CLEAR(TheTable);
TheTable.LOCKTABLE;
```

There are 3 ways to loop the records:

Method 1:

// If you only need to change 1 field and you don’t need to use VALIDATE and you don’t care if someone else has changed the record between the moment you read it and now that you will change it, you can use MODIFYALL.

```
tmpTheTable.RESET;
IF tmpTheTable.FINDSET THEN
  REPEAT
    TheTable.RESET;
    TheTable.SETRANGE(…) // on all primary key fields
    TheTable.MODIFYALL("Some Field",tmpTheTable."Some Field");
  UNTIL tmpTheTable.NEXT = 0;
```

Method 2:

```
// This code uses the version of each record in the temptable to
change the record in the table. If one of the records has been
changed in the meantime, it will generate an error. (remark: the
MODIFY will also generate a SELECT EXCLUSIVE-LOCK before the UPDATE-
statement. This is necessary because NAV has to check if the version
of the record is still the same.)
tmpTheTable.RESET;
IF tmpTheTable.FINDSET THEN
  REPEAT
    TheTable := tmpTheTable;
    TheTable."Some Field" := tmpTheTable."Some Field"; // or use
VALIDATE
    TheTable.MODIFY(FALSE);
  UNTIL tmpTheTable.NEXT = 0;
```

Method 3:

```
// This code retrieves the record again (with EXCLUSIVE-LOCK!) and
then changes it. If the record was changed between the first read and
now, it will NOT generate an error on the MODIFY because you will
have received the latest version.
tmpTheTable.RESET;
IF tmpTheTable.FINDSET THEN
  REPEAT
    CLEAR(TheTable);
    TheTable := tmpTheTable;
    TheTable.FIND('=');  // or TheTable.GET(primary key). Both
commands are possible but you need to be sure you don't have filters
on "TheTable"
    TheTable."Some Field" := tmpTheTable."Some Field"; // or use
VALIDATE
    TheTable.MODIFY(FALSE);
  UNTIL tmpTheTable.NEXT = 0;
```

- You change a record and each time you get an error that someone else changed the record but there is no one else in the DB.
  This is an error that I see happening a lot. And no, it is not a bug in NAV, but a bug in your C/AL code. Basically what happens:
  - In your main code, you get a record A. Now this record is "version 1" in both your variable and in the DB.
  - You pass this record "version 1" to a function WITHOUT specifying the VAR=true property. This means NAV creates a copy of your variable and in the function you work with the copy (also "version 1"). Let's call this parameter record "version 1 bis".
  - In that function you modify the variable and save it to the database. So "version 1 bis" is changed to "version 2" and saved to the DB. In the DB and your parameter variable you now have now "version 2".

    - You leave the function and go back to your main function. The variable still has "version 1"!!!!
    - In your main function (or in another function called after the first function) change the record and want to save it to the DB. When you run the MODIFY-statement, NAV checks if the version is still the same. Your variable is "version 1" but in the database it finds "version 2" => ERROR : someone updated your variable!

  How to fix it?

    - You can use your parameters with the VAR=TRUE parameter. This way NAV does not copy your variable but it is the SAME variable. It has another name, but both point to the same bytes in your memory.
    - Another possibility: In case you don't pass the variable to the first function but just do a get inside. Just after coming back from that first function, get the record back from the DB. You will then have the latest version.

- Deleting a record
    - What is there to do better? Get the record and then delete it!
      Actually, in certain circumstances you can do it more efficient.
      Like this :

      ```
      IF TheTable.GET(…) THEN
        TheTable.DELETE(FALSE);
      ```

      You can do this better. You don't need to get the record first. Just put filters and do a DELETEALL(FALSE).
      You don't need first to see if the record exists and if it exists, delete it. The DELETEALL handles this all.
      Of course, if you need to run the OnDelete-trigger, NAV will need to do a select anyway, but it is still faster, because it is all internal code, no C/AL code is involved (in PRE-NAV2013) and probably also faster in NAV2013+ because the C# executed is behind the scenes and not converted from C/AL which, the first time executed, still has to be compiled.

      ```
      TheTable.RESET;
      TheTable.SETRANGE(..); // set filters on all the primary key fields
      TheTable.DELETEALL(FALSE);
      ```

    - I also see regularly this code:

      ```
      TheTable.RESET; // and sometimes not even a RESET…
      TheTable.SETRANGE/SETFILTER
      IF TheTable.FINDSET THEN // or FIND('-')
        REPEAT
          TheTable.DELETE(FALSE); // or DELETE(TRUE);
        UNTIL TheTable.NEXT = 0;
      ```

      Way better (and shorter) is this:

      ```
      TheTable.RESET; // and sometimes not even a RESET…
      TheTable.SETRANGE/SETFILTER
      TheTable.DELETEALL(FALSE); // or DELETEALL(TRUE);
      ```

- Inserting records
    - How can we improve the performance for inserting?
      Depending on some factors, this can be improved. Inserting a single record can't be improved a lot, but if you need to insert a lot of records it can in certain cases.
    - What is the improvement?
      The improvement is leveraging the bulk insert of NAV (this is completely different from the bulk insert of SQL!).

  - What is the NAV bulk insert?
    - In general, when something has to be written to the DB (also a delete is writing to the DB!), NAV sends it immediately to the DB, starting a transaction at the very first record and keeps that transaction open until the very last record when the complete C/AL code finishes working or when a COMMIT is issued.
    - With the bulk insert, NAV does NOT immediately send the records to be inserted to the DB but buffers them up until any other action (reading or writing except for a bulk insert) on that table happens or until when the buffer is full. In that moment, NAV sends all records in one go to SQL Server. This is a lot faster than playing ping pong with the server.
    - Like the name implies, ONLY INSERT can be done in bulk. MODIFY,DELETE are immediately send to the server.
    - What are the conditions to use bulk insert?
      - You don't have any direct control over it, but to use bulk insert, these conditions must be true:
        - Only TheTable.INSERT(FALSE);
          - So no :
            - IF TheTable.INSERT(FALSE) THEN
            - theTable.INSERT(TRUE); // if there is code in the OnInsert-trigger
        - The table does not have a field with property AutoIncrement=TRUE
        - The table must not contain BLOB fields
        - PRE-NAV2013
          - The table must not contain Variant or RecordID fields
        - NAV2013+
          - In the service config-file, value "BufferedInsertEnabled" must be TRUE.
    - Bulk insert can lead to an "Record already exists" error. See later.
- Renaming records
  - Avoid it whenever possible!
    It will run through the tables to change the values. It can take a long time and cause a lot of blocking.

VARIA

- "Record already exists" pops up on a GET/FIND/MODIFY/… statement but not on the INSERT itself
  - In this case you have been bitten by a bug in your C/AL code that is using the bulk insert.
    What is happening?
    You are inserting a lot of records and some of the records is inserted twice (in the same loop of inserts or it already existed in the DB). Why didn't it pop up on the INSERT-statement?
    That is because the bulk insert feature delayed inserting the incriminating record until another action was done on that table.

- Reading/inserting/modifying/deleting records in another company than the current one.

  - IMPORTANT: Remember that ONLY that record will be in the other company. Any other record-variable accessed by variables in the code in that record or in the Tablerelation of a field will be on the CURRENT company.
  - (VERY) WRONG: you should NEVER use triggers when inserting/modifying/deleting a record that is in another company. All the code behind the triggers/Tablerelations will work on the CURRENT company.

```
CLEAR(TheTable);
TheTable.CHANGECOMPANY('Some Other Company');
TheTable.VALIDATE("Field 1",'Some Value');
TheTable.VALIDATE("Field 2",'Some Value');
TheTable.VALIDATE("Field 3",'Some Value');
TheTable.INSERT(TRUE);
```

  - CORRECT: to insert a record into another company (MODIFY and DELETE work in the same way).

```
CLEAR(TheTable);
TheTable.CHANGECOMPANY('Some Other Company');
TheTable."Field 1" := 'Some Value';
TheTable."Field 2" := 'Some Value';
TheTable."Field 3" := 'Some Value';
TheTable.INSERT(FALSE);
```

  - CORRECT: Reading 1 record in another company

```
CLEAR(TheTable);
TheTable.CHANGECOMPANY('Some Other Company');
TheTable.GET(....);
```

  - CORRECT: Reading/deleting/modifying 1 or multiple record(s) in another company

```
CLEAR(TheTable);
TheTable.CHANGECOMPANY('Some Other Company');
TheTable.SETRANGE("Field 1",'Some Value');
TheTable.FINDSET;
REPEAT

UNTIL TheTable.NEXT = 0;
```

Or

```
TheTable.DELETEALL(FALSE); // No triggers!!!!!
```

Or

```
TheTable.MODIFYALL(FALSE); // No triggers!!!!!
```

- If you need 2 or more recordvariables of the same record, you can also use an array instead of 2 variables. The 2 variables work independently of each other like using 2 variables. Some examples.
    - RecordVar1.RESET; => RecordVar[1].RESET;
    - CLEAR(RecordVar1); => CLEAR(RecordVar[1]);
    - RecordVar1.SETRANGE(... => RecordVar[1].SETRANGE(…
    - CLEAR(RecordVar1); CLEAR(RecordVar2); … => CLEAR(RecordVar);
  - You can also use a variable to indicate the element. An example:

    SomeInteger := 1;
    RecordVar1.RESET; => RecordVar[SomeInteger].RESET;

- The array technique can also be used on temporary record-variables. They work also independently of each other like on a real table. BUT THERE IS ONLY 1 TEMPTABLE! This means that if you create a record with element [1], it will be available in element [2]! An example (This is the very first code on the temptable, so there are no records in it!).

```
CLEAR(recRecord[1]);
recRecord[1]."No." := '1');
```

```
recRecord[1].INSERT(FALSE);
recRecord[2].RESET;
recRecord[2].FINDFIRST;
MESSAGE('%1',recRecord[2]."No."); // this will show the value of the
record I created in element [1]!
```

- Make a total amount per "Gen. Bus. Posting Group" and "Gen. Prod. Posting Group" of table 15:"G/L Entry".
    - In SQL it would be easy:

      ```
      SELECT "Gen_ Bus_ Posting Group","Gen_ Prod_ Posting
      Group",sum(Amount)
           FROM dbo."CRONUS International Ltd_$G_L Entry"
           GROUP BY "Gen_ Bus_ Posting Group","Gen_ Prod_ Posting Group"
           ORDER BY "Gen_ Bus_ Posting Group","Gen_ Prod_ Posting Group"
      ```
    - PRE-NAV2013 and NAV2013!!!!!!!
        - But in C/AL we can't use SQL-statements (or we must use ADO), so there is another way. I advise to always this way to do some summing in C/AL.

          ```
          GLEntry.RESET;
          GLEntry.SETCURRENKEY(...); // try to get a key that can be used
          to find the filtered data in a fast way
          GLEntry.SETRANGE(.....); // put your filters
          IF GLEntry.FINDSET then
            REPEAT
              // "GLEntryTMP" is the temptable we will use to store the
          grouping totals
              GLEntryTMP.RESET;

              // try to get a good key for the filters
              IF NOT GLEntryTMP.SETCURRENTKEY("Gen. Bus. Posting Group")
          THEN
                IF NOT GLEntryTMP.SETCURRENTKEY("Gen. Prod. Posting
          Group") THEN ;

              // I filter on the records for which I want to group the
          records
              GLEntryTMP.SETRANGE("Gen. Bus. Posting Group",GLEntry."Gen.
          Bus. Posting Group");
              GLEntryTMP.SETRANGE("Gen. Prod. Posting Group",GLEntry."Gen.
          Prod. Posting Group");

              IF NOT GLEntryTMP.FINDFIRST THEN BEGIN
                // I didn't find a record, so I have to create a new one.
                // Remember that to insert a record, you have to respect a
          unique primary key.
                //   This is done, because EACH "GLEntry" I read is
          unique, so I can just insert
                //     "GLEntry" in my temptable.
                GLEntryTMP := GLEntry;
                GLEntryTMP.INSERT(FALSE);
              END
              ELSE BEGIN
                // I found the record with combination I wanted, so I add
          the field(s) I am SUMming
                GLEntryTMP.Amount += GLEntry.Amount;
                GLEntryTMP.MODIFY(FALSE);
              END;
            UNTIL GLEntry.NEXT = 0;

          // In the temptable I know have records that contain the SUM for
          the combination I wanted.
          // Of course if you want to sort your data in a certain way, you
          will need a key for it and with SQL you can put property
          MaintainSQLIndex to FALSE.
          // Another way is to create a new table with all the fields and
          keys you need and use that as
          ```

```
//   temptable. You DON'T need to license a new table that you
use ONLY as temptable!

// Now you can just read the temptable and do what you want to
do with your totals
GLEntryTMP.reset;
FORM.RUNMODAL(0,GLEntryTMP);
```

- NAV2013+
  - Using a Query-object
    - This is the closest we can get to a pure SQL-statement.
    - Some things you need to keep in mind:
      - Because you have a limited amount of fields, you can use SQL covering indexes. A covering index is an index that contains all fields in the statement. So in this case : create a NAV key with fields "Gen. Bus. Posting Group", "Gen. Prod. Posting Group", "Amount". That means that SQL can just read the index and doesn't need to get the record. This can speed up the query a lot.
      - It is even possible to speed it up even more if you create a key with fields "Gen. Bus. Posting Group", "Gen. Prod. Posting Group" and SIFT = "Amount". You can put property "MaintainSQLIndex" to False because you don't need it for this query. NAV will see that it can use the SIFT for the query and query the SIFT instead of the table. This will even speed up the query even more.
  - We can still use pure C/AL. But it gets even better: Check the differences. Only for NAV2013R2 and later. **NOT FOR NAV2013!!!!!!**

```
GLEntry.RESET;
~~GLEntry.SETCURRENKEY(...); // try to get a key that can be used
to find the filtered data in a fast way~~
GLEntry.SETRANGE(.....); // put your filters
IF GLEntry.FINDSET then
  REPEAT
    // "GLEntryTMP" is the temptable we will use to store the
grouping totals
    GLEntryTMP.RESET;

~~    // try to get a good key for the filters~~
~~    IF NOT GLEntryTMP.SETCURRENTKEY("Gen. Bus. Posting Group")
THEN~~
~~      IF NOT GLEntryTMP.SETCURRENTKEY("Gen. Prod. Posting
Group") THEN ;~~
    // Remark: testing with or without SETCURRENTKEY with
NAV2016 gave me inconclusive results to what is more performant.
Personally, I won't use it anymore.

    // I filter on the records for which I want to group the
records
    GLEntryTMP.SETRANGE("Gen. Bus. Posting Group",GLEntry."Gen.
Bus. Posting Group");
    GLEntryTMP.SETRANGE("Gen. Prod. Posting Group",GLEntry."Gen.
Prod. Posting Group");

    IF NOT GLEntryTMP.FINDFIRST THEN BEGIN
      // I didn't find a record, so I have to create a new one.
      // Remember that to insert a record, you have to respect a
unique primary key.
      //   This is done, because EACH "GLEntry" I read is
unique, so I can just insert
      //     "GLEntry" in my temptable.
      GLEntryTMP := GLEntry;
      GLEntryTMP.INSERT(FALSE);
    END
    ELSE BEGIN
```

```
      // I found the record with combination I wanted, so I add
the field(s) I am SUMming
      GLEntryTMP.Amount += GLEntry.Amount;
      GLEntryTMP.MODIFY(FALSE);
    END;
  UNTIL GLEntry.NEXT = 0;

// In the temptable I know have records that contain the SUM for
the combination I wanted.
~~// Of course if you want to sort your data in a certain way, you
will need a key for it and with SQL you can put property
MaintainSQLIndex to FALSE.~~
// if you want to sort your data in a certain way, you don’t
need a key anymore!
// Another way is to create a new table with all the fields and
keys you need and use that as
//   temptable. You DON'T need to license a new table that you
use ONLY as temptable!

// Now you can just read the temptable and do what you want to
do with your totals
GLEntryTMP.reset;
FORM.RUNMODAL(0,GLEntryTMP);
```

- Something on creating secondary keys in NAV
  - PRE-NAV2013
    - Each SETCURRENTKEY needs a key. BUT you don’t need to maintain the key in SQL Server.
  - NAV2013+
    - SETCURRENTKEY does not need a key (except NAV2013 for which it is necessary…)
  - General rules for keys:
    - Put the most selective fields first (=if filtered on those, you exclude the most possible records)
      - So "Document Type","Document No." is not so good. Better is "Document No.","Document Type". The reason this is not done is because the key is used to show multiple document numbers in the list.
      - Put fields where you filter on equality (=SETRANGE("xxx",'1 value')) first and then fields filtered on inequality (= using SETFILTER)
    - It is not always necessary to create a new key to speed up your query. An existing key might be "good enough".
    - New key : reading will be faster (if your key is ok…) but writing to the table will be slower! So sometimes it is better not to create a new key.
    - SIFT: NAV will take the first SIFT that has all the fields to filter on and has the necessary totals. But that might mean that it needs to scan the SIFT-view.
      - So if you have:
        - "Document Type","Document No." with SIFT "Quantity".
        - "Document No.","Document Type" with SIFT "Quantity".
        - And you filter only on "Document No.", NAV will take the first key and performance can be bad because it will need to scan the whole SIFT-table/SIFT-view. You should also filter on "Document Type" OR invert the order of the 2 keys OR disable maintaining the SIFT of the first key.