Tips and tricks with kbmMemTable

kbmMemTable is an in memory row/column oriented database which have a vast number of features and which is very fast in both storing data and locating stored data. I will in this article focus on some basic things and on some of the less known, but very useful, features of kbmMemTable.

Create a memory table with indexes in code

```
var
  mt:TkbmMemTable;
begin
  // Define fields (you may already have them defined at designtime).
  mt.FieldDefs.Add('ID',ftInteger);
  mt.FieldDefs.Add('Name',ftString,80);
  mt.FieldDefs.Add('Address',ftString,80);

  // Define indexes (if you have a large amount of records in your mt
  // indexes will make searches faster, but inserts/edits slower).
  mt.IndexDefs.Add('iID','ID',[ixPrimary]);
  mt.IndexDefs.Add('iName','Name',[ixCaseInsensitive]);
  mt.CreateTable;
  mt.Open;
end;
```

When searching on for records using the Name field as search criteria, we have defined that it should search case insensitively, so for example uppercase A is the same as lowercase a.

Fast insertion into the database

kbmMemTable exists in two primary versions, Standard Edition and Professional Edition. Standard Edition can be purchased separately, while Professional Edition only is available as a bundle with kbmMW Professional or kbmMW Enterprise Edition.

While kbmMemTable Professional is the absolutely fastest memory table in the world, kbmMemTable Standard Edition can almost reach its performance when used correctly.

If you are to insert a large number of records in kbmMemTable Standard Edition, then you will want to do like this:

```
var
  fldID,
  fldName,
  fldAddress:TField;
begin
  // First get quick access to the fields.
  // This prevents having to search for a field each
  // and every time a record is to be inserted.
  fldID:=mt.FieldByName('ID');
  fldName:=mt.FieldByName('Name');
  fldAddress:=mt.FieldByName('Address');

  // Prevent update of attached controls on each insert.
  // This is a big performance factor when inserting huge
  // amounts of records.
  mt.DisableControls;

  // Prevent update of indexes on each insert. Professional
  // Edition is much faster in on the fly updates of indexes,
  // so it will perform extremely fast even without disabling
  // indexes while inserting. But the highest performance
  // is obtained by disabling them before the batch insert.
  mt.EnableIndexes:=false;

  // Insert a lot of records.
  for i:=1 to 1000000 do
  begin
    mt.Append;
    fldID.AsInteger:=i;
    fldName.AsString:='Name'+inttostr(i);
    fldAddress.AsString:='Address'+inttostr(i);
    mt.Post;
  end;

  // Enable and update indexes.
  mt.EnableIndexes:=true;
  mt.UpdateIndexes;

  // Enable updating attached controls (grids etc).
  Mt.EnableControls;
```

How to search?

You either choose to switch to the index representing the column(s) you want to search on, or you simply just search and let kbmMemTable find a relevant index (if any) to use for the search.

If you want the absolutely fastest result, then you should switch to the index first, to avoid that kbmMemTable have to make an additional search on your current index to sync with the found place in the search index.

However the second index sync search is generally very fast.

You can use Locate to search for a complete name for example. If it finds a record with the given name, that record will be the current record, and you can access other fields in it immediately.

```
if mt.Locate('Name','Jens Hansen',[]) then
   MessageDlg('Found record. Address='+mt.FieldByName('Address').AsString);
```

You can use Lookup to search for a record containing the given value, and return the contents of any field as result. It will not move the current record.

```
v:=mt.Lookup('Name','Jens Hansen','Address');
if not VarIsNull(v) then
   MessageDlg('Found record. Address='+v);
```

You can also search for partial values

```
if mt.Locate('Name','Jens',[loPartialKey]) then
   MessageDlg('Found record. Address='+mt.FieldByName('Address').AsString);
```

How to search on multiple fields?

```
if mt.Locate('ID;Name',[1,’Jens Hansen'],[]) then
   MessageDlg('Found record. Address='+mt.FieldByName('Address').AsString);
```

Let kbmMemTable automatically create indexes for you to improve search performance

```
mt.AutoAddIndexes:=true;
```

This may however result in creation of many indexes over time if you often search on new columns and having many indexes to maintain when inserting/editing/deleting records will always give a performance penalty, so use with care.

How to sort?

Adding an index will always result in the data being sorted according to an index. So you can add an index and switch to it.

```
mt.AddIndex('iMyIndex','Name',[ixDescending]);
mt.IndexName:='iMyIndex';
```

Or even easier:

```
mt.SortOn('Name',[mtcoDescending]);
```

How to only show certain records?

For this purpose, you can use a range or a filter.

A range can limit the records displayed in for example a grid, to show only records within a range (eg. 10<=ID<=20, only show records where the ID is between 10 and 20). The fields participating in the range must be part of an index.

```
mt.IndexName:='iID';
mt.SetRange([10],[20]);
```

This will use the index iID, and setup a range to only make records having ID in the range 10 to 20 (inclusive) available.

Another variant exists of the SetRange method, which accepts a string which contains names of fields that is to be filtered on. A new range index will automatically be generated, and switched to.

```
mt.SetRange('ID',[10],[20]);
```

Cancel a range again by

```
mt.CancelRange;
```

Delete all records within a range

```
mt.DeleteRange('ID',[10],[20]);
```

Filter records using a filter expression

```
mt.Filter:='(ID>=10) AND (ID<=20)';
mt.Filtered:=true;
```

This will give the same result as using the range above, but do not require any indexes. It however scans every record to find matching records, and will thus be slow on large datasets.

The advantage with the expression filter is that you can make quite complex filters using AND/OR.

When a field name contains spaces, you must enclose the field name in brackets. For example:

```
[Home State] = 'CA' or [Home State] = 'MA'
```

The FilterOptions property controls case sensitivity and filtering on partial comparisons.

The Filter can be changed at runtime at wish.

The following field types can be part of a filter:

ftSmallInt, **ftWord**, **ftInteger**, **ftAutoInc**, **ftFloat**, **ftCurrency**, **ftString**, **ftFixedChar**, **ftDate**, **ftTime**, **ftDateTime**, **ftBoolean**, **ftBCD**, and any other field which can return a string or a numeric value.

The following operators are available:

Arithmetic:

+ Addition can be used on all numeric and string expressions.
- Subtraction can be used on all numeric expressions.
***** Multiplication can be used on all numeric expressions.
/ Division can be used on all numeric expressions.

Conditions:

= Equal
> Greater than
< Less than
>= Greather than or equal

<= Less than or equal
<> Not equal
IS NULL True if expression is null
IS NOT NULL True if expression is not null
NOT Negates the boolean expression
OR True if one of the two boolean expressions are true
AND True if both the two boolean expressions are true
IN (..., ...) True if the result of the left side expression is found in the list of values.
LIKE '....' True if the left side string expression is matching the wildcards in the right side string.
The wildcards consists of:
***** Matches any number of unknown characters.
? Matches exactly one unknown character.

An example: Fld1 LIKE 'JOHN*'

Functions:

UPPER(expression **)** Convert string expression to uppercase.
LOWER(expression **)** Convert string expression to lowercase.
SUBSTRING(expression **,** startpos **,** count **)** Extract a substring from the expression string starting at startpos and with max length of count chars. The count parameter can be omitted, in which case is means the rest of the string.
TRIM(expression **)** Trim string expression for all leading and trailing spaces.
TRIMLEFT(expression **)** Trim string expression for all leading spaces.
TRIMRIGHT(expression **)** Trim string expression for all trailing spaces.
GETDATE Returns a floating point date/time value for current time.
YEAR(expression **)** Return the year of a floating point date/time value.
MONTH(expression **)** Return the month of the year (1..12) of a floating point date/time value.
DAY(expression **)** Return the day of the month (1..28/29/30/31) of a floating point date/time value.
HOUR(expression **)** Return the hour (0..23) of a floating point date/time value.
MINUTE(expression **)** Return the minute (0..59) of a floating point date/time value.
SECOND(expression **)** Return the second (0..59) of a floating point date/time value.
DATE(datestring **,** formatstring **)** Convert the datestring to a floating point date/time value according to the SysUtils.FormatDateTime format string .
DATE(expression **)** Return the date part of a floating point date/time value.
TIME(timestring **,** formatstring **)** Convert the timestring to a floating point date/time value according to the SysUtils.FormatDateTimeformat string .
TIME(expression **)** Return the time part of a floating point date/time value.

Filtering however is done on each record every time that record is accessed, thru scrolling or counting number of records etc. So it's fairly slow on large datasets to use a filter.

A better way, is to create a filtered index. That is a real index which only contains references to records that live up to the filter expression.

```
mt.AddFilteredIndex('iFilteredIndex','ID',[],'(ID>=10) AND (ID<=20)',[]);
```

This creates a new index, ordered (and quickly searchable) by ID, and filtered by the given filter string. You can filter on any field or calculation of fields. It doesn't have to be fields that are in the key fields list.

Every time you add or alters records, the index will be updated and records may this way disappear or appear in the filtered index depending on the values of the record.

When you switch to a filtered index, and thus only show the records participating in that index, scrolling through or searching isjust as fast as if you had no filter defined. Inserts and updates have a small performance penalty, so if you have many records to insert, use the batch insert method outlined earlier in this article.

"Standard" indexes?

With standard indexes, I refer to indexes which are created/defined by the memtable during its course of operation.

When a memory table is opened, there always exist one single index, the row order index. It's the index that is updated to ensure that order of insertion of records can be established. It's also responsible for holding a reference to all records. The row order index is selected by setting the IndexName to an empty string.

Internally you will find the roworder index to be named '__MT__ROWORDER_'. A string constant exists for that name: kbmRowOrderIndex.

The row order index can't be deleted.

Further a number of other indexes may be defined at various occations.

- '__MT__DETAIL_' (kbmDetailIndex). This index is created (and automatically recreated) when the master record changes in a master/detail relationship. It ensures that browsing through detail records is very fast, and quick to search on.
- '__MT__DEFSORT_' (kbmDefSortIndex). This index is created when you are using the Sort or SortOn methods. It is only recreated when you issue another Sort/SortOn method call.
- '__MT__DEFAULT_' (kbmDefaultIndex). This index is automatically created when you set IndexFieldNames to a set of fields for which no index already exist.
- '__MT__RANGE_' (kbmRangeIndex). This index is automatically created when you apply a range filter.
- '__MT__AUTO_' (kbmAutoIndex). This is actually not only one index, but potentially many indexes, which are automatically created if you have the property AutoAddIndexes set to true, and you make a search (locate/lookup) on fields for which there are no existing indexes available. The actual indexname will be '__MT__AUTO_' + the semicolon separated field names. Be aware that this index is not automatically destroyed, so if you search on many weird combinations of fields, you may end up with many indexes, and may need to delete them yourself if needed.

Index inheritance

If you want to temporarily sort on a different field than the ones participating on the current selected index (it could for example be a filtered index), you simply do SortOn, as described earlier on, while having the filtered index selected. This will create a new sorted index, based on the filtered index, so only records that are acceptable in the base index, are candidate for inclusion in the new sort index.

Transactions and versioning

If you make modifications to a number of records in a table, it would, in some situations, be nice to be able to roll back those changes in one go. For example if you have an application where the user can make modifications to a number of entries (inventory for example), and at a later stage decide to save those changes or cancel them. Cancelling the changes would require a roll back, while saving means that the user commits to his changes.

kbmMemTable supports whats called transaction management via the methods StartTransaction, Commit and Rollback. Further it also supports an Undo function on the record level. More about that in a moment.

To use transaction management, one also need to understand versioning.

kbmMemTable have two properties controlling versioning: **EnableVersioning** and VersioningMode. **EnableVersioning** is a Boolean, while **VersioningMode** can have the value of **mtvm1SinceCheckPoint** or **mtvmAllSinceCheckPoint**.

Versioning allows kbmMemTable to store multiple versions of the same record. Whenever a record is inserted into the memory table, one version of that record exists, the one that was inserted.

Default **EnableVersioning** is **false** and thus the records you see in the memory table are the ones you have. No other versions are created. If **EnableVersioning** is set to **true**, then kbmMemTable will start to make multiple versions of a record, depending on the setting of **VersioningMode**.

If **VersioningMode** is set to **mtvm1SinceCheckPoint**, then at most, an original version is kept, and the altered one. This is cool, if you need to update an external storage with the changes made in the memory table. Then the original version identifies what you have to search for, and the new version identifies what to change to.

A *deltahandler* is what is used for handling those scenarios, but now we will focus on transactions. So in this case, you can only undo back to the original version, and not intermediate changes.

Example:

```
mt.AppendRecord(…);
mt.StartTransaction;
mt.Edit….mt.Post;
mt.Delete;
mt.Rollback;
```

In this example a record is inserted, a transaction is started, then the record is edited and then deleted. With **VersioningMode** set to *mtvm1SinceCheckPoint*, then you can only choose to roll back to where the record was inserted. The version containing the edited record is lost the moment the record is deleted. Only one version in addition to the original record is kept. To allow for a multi level undo, then **VersioningMode** should be set to **mtvmAllSinceCheckPoint**. Then you can do this:

```
mt.AppendRecord(…);
mt.StartTransaction;    // First level of transaction
mt.Edit….mt.Post;
mt.StartTransaction;    // Second level of transaction
mt.Delete;
mt.Rollback;            // Back to the edited record again
mt.Rollback;            // Back to the original appended record
```

You see you can nest transactions this way. If you want to make a change, that is protected by a StartTransaction, stick, then you call mt.Commit. After the commit, the change can only be rolled back if there are multiple levels of nested transactions as in this example:

```
mt.AppendRecord(…);
mt.StartTransaction;     // First level of transaction
mt.Edit….mt.Post;
mt.StartTransaction;     // Second level of transaction
mt.Delete;
mt.Commit;               // We will keep the delete but not the edited record.
mt.Rollback;             // Ah no.. we will revert to the original appended record
```

You can check the state of a record by checking the UpdateStatus property. It returns the current records state. Was it inserted (**usInserted**), deleted (**usDeleted**), modified (**usModified**) and unchanged (**usUnmodified**). To actually see **usDeleted** marked records, it require adding and switching to a special index that allows for showing deleted records… check the overloaded version of **AddIndex** which accepts providing a set of UpdateStatus flags, for which records to include in the index.

When you insert/append records, those will be marked with **usInserted** as their **UpdateStatus** propery. That also happens when you load the records from another dataset, because technically they have been inserted into the memory table. You can however ask kbmMemTable to consider these newly loaded records, as being the original ones and thus that they should have the **UpdateStatus** of **usUnmodified**.

```
mt.CheckPoint;
```

The latest version of any record is now considered the original version and marked as **usUnmodified**, and all intermediate versions have been removed. That also means that if records were deleted, they are now permanently deleted and can’t be recovered.

To undo last change of the current record, you can use:

```
mt.Undo;
```

Remember that the number of undo’s you can make on the current record, depends on the setting of **EnableVersioning** and **VersioningMode**.

The deltahandler

A deltahandler is a class that can scan through a memory table, and determine what has happened with each record, and allow the developer to do something depending on what happened. Was it inserted, deleted, modified or was nothing changed with it?

It is essentially a great tool for making changes in your dataset resolve back to from where the data originated from (typically a SQL database or files in a file system etc).

This is a simple deltahandler:

```
// An example on how to create a deltahandler.
TMyDeltaHandler = class(TkbmCustomDeltaHandler)
protected
   procedure InsertRecord(var Retry:boolean; var State:TUpdateStatus); override;
   procedure DeleteRecord(var Retry:boolean; var State:TUpdateStatus); override;
   procedure ModifyRecord(var Retry:boolean; var State:TUpdateStatus); override;
//   procedure UnmodifiedRecord(var Retry:boolean; var State:TUpdateStatus); override;
end;

procedure TMyDeltaHandler.InsertRecord(var Retry:boolean; var State:TUpdateStatus);
var
   i:integer;
   s,sv:string;
   v:variant;
begin
     s:='';
     for i:=0 to FieldCount-1 do
     begin
          v:=Values[i];
          if (VarIsNull(v)) then
             sv:='<NULL>'
          else if not (Fields[i].DataType in kbmBinaryTypes) then
             sv:=v
          else
             sv:='<Binary data>';
          s:=s+sv+' ';
     end;
     ShowMessage(Format('Inserted record (%s)',[s]));
end;

procedure TMyDeltaHandler.DeleteRecord(var Retry:boolean; var State:TUpdateStatus);
var
   i:integer;
   s,sv:string;
   v:variant;
begin
     s:='';
     for i:=0 to FieldCount-1 do
     begin
          v:=Values[i];
          if (VarIsNull(v)) then
             sv:='<NULL>'
          else if not (Fields[i].DataType in kbmBinaryTypes) then
             sv:=v
          else
             sv:='<Binary data>';
          s:=s+sv+' ';
     end;
     ShowMessage(Format('Deleted record (%s)',[s]));
end;

procedure TMyDeltaHandler.ModifyRecord(var Retry:boolean; var State:TUpdateStatus);
var
   i:integer;
   s1,s2,sv:string;
   v:variant;
begin
     s1:='';
     s2:='';
     for i:=0 to FieldCount-1 do
     begin
          v:=Values[i];
          if (VarIsNull(v)) then
             sv:='<NULL>'
          else if not (Fields[i].DataType in kbmBinaryTypes) then
             sv:=v
          else
             sv:='<Binary data>';
          s1:=s1+sv+' ';

          v:=OrigValues[i];
          if (VarIsNull(v)) then
             sv:='<NULL>'
          else if not (Fields[i].DataType in kbmBinaryTypes) then
             sv:=v
          else
             sv:='<Binary data>';
          s2:=s2+sv+' ';
     end;
     ShowMessage(Format('Modified record (%s) to (%s)',[s2,s1]));
end;

//procedure TMyDeltaHandler.UnmodifiedRecord(var Retry:boolean; var State:TUpdateStatus);
//begin
//end;
```

And you start the deltahandler like this:

```
var
   myDeltaHandler:TMyDeltaHandler;
begin
   myDeltaHandler:=TMyDeltaHandler.Create(nil);
   try
       mt.DeltaHandler:=myDeltaHandler;
       mt.Resolve;
   finally
      mt.DeltaHandler:=nil;
      myDeltaHandler.Free;
   end;
end;
```

Loading data

Data can be loaded into a memory table in multiple ways:

1. Inserting them one record at a time
2. Loading the data from another dataset
3. Loading the data from a file

As for 1, you can use the ordinary Insert/Post, or AppendRecord methods.

As for 2, you can use:

```
mt.LoadFromDataset(anotherdataset, [mtcpoStructure]);
```

This will make your memory table have the same fields as the anotherdataset (due to the **mtcpoStructure** option), and load a copy of all data from the dataset into the memory table.

A number of copy options exists:

mtcpoStructure: Copies table structure (field definitions) from the source. Any field definitions you had in your local memory table are removed.

mtcpoOnlyActiveFields: Only field definitions in the source that actually are represented as a true field, are copied.

mtcpoProperties: Copy over field properties like DisplayWidth, DisplayLabel, Required, ReadOnly, Visible, DefaultExpression, Alignment, ProviderFlags, Lookup, LookupCache, LookupDataset, LookupKeyFields, LookupResultField, KeyFields, DisplayFormat, EditFormat, MaxValue, MinValue, DisplayValues, TransLiterate, Precision, Currency and BlobType.

mtcpoLookup: Also copy lookup fields from the source dataset.

mtcpoCalculated: Also copy calculated fields from the source dataset.

mtcpoAppend: Append records to the existing records in the memory table. This cant be used with **mtcpoStructure** or mtcpoProperties.

mtcpoFieldIndex: Copy the index position of fields to ensure field order is the same as in the original.

mtcpoDontDisableIndexes: Default a batch insertion is made, where indexes are only updated after all records have been copied over. However if you for example have an index with unique constraint on a field, then you might want to have your copy to stop early, if there is a duplicate of that field value. That require that the indexes are updated on the fly for each record. Standard Edition will have a larger performance penalty than Professional Edition of kbmMemTable for this situation.

mtcpoIgnoreErrors: Ignore any copy errors instead of stopping copying.

mtcpoLookupAsData: Copy over lookup fields a regular datafield, with its matching data.

mtcpoCalculatedAsData: Copy over calculated fields as a regular data field with its calculated data.

mtcpoStringAsWideString: Assume that source ftString, ftFixedChar fields should be created as ftWideString in your local memory table.

mtcpoWideStringUTF8: If string/character/memo fields in the source dataset do not match "wideness" of the same in the destination dataset (your local memory table), then automatically encode or decode to/from UTF8. Eg. If the source dataset has field 'str1' which is a ftWideString field, while the local memorytable has the same field 'str1' defined as a ftString field, it will UTF8 encode the data coming from the source field before putting it into the destination field. Similarly if a source field is of type ftString, but the destination is of type ftWideString then an UTF8 decoding will take place before putting the value into the destination field. The same takes place with ftMemo/ftWideMemo and ftFixedChar/ftWideFixedChar.

If you already have an existing field structure in your local memory table, and want to copy fields from a source dataset, where the field names do not match, you can take advantage of field name mapping.

Eg.

```
mt.LoadFromDataset(anotherdataset, [],'str1=str_1;int2=int_2');
```

This is telling kbmMemTable that the local field named str1 is called str_1 in the source table etc.

As for option 3, loading from a file, you can use:

```
mt.LoadFromFileViaFormat('somefilename',someformatinstance);
```

someformatinstance is an instance of a stream format component. kbmMemTable comes with two, **TkbmMemBinaryStreamFormat** and **TkbmMemCSVStreamFormat**.

The binary stream format class stores data compactly and efficiently, and is giving the fastest performance. However the CSV stream format class allows you to read in (and write out) comma separated formatted data, for easy integration with other external systems. kbmMW (our middleware product) comes with additional stream formatters for XML and JSON.

Each stream format instance have a number of flags that can be set, to tell how it's supposed to handle the reading or writing of data, but that is a story for another time.

```
sf:=TkbmMemCSVStreamFormat.Create(nil);
try
   mt.LoadFromFileViaFormat('.\mydata.csv',sf)
finally
   sf.Free;
end;
```

Extracting data

A number of methods exist for easily extract data from a memory table.

```
mt.Extract('fld1;fld2',slData);
```

This one will extract the fields fld1 and fld2 for all records to a TStrings instance (TStringList typically) that you will have to create beforehand. If the optional AFormat string is given, each line in TStrings will be formatted according to that format. If none is given, then all field values for a record will be separated by a space. The above example will thus put a space between the value of fld1 and fld2 for each record/line.

```
mt.Extract('fld1;fld2',slData,'%s=%s');
```

This will put an equal sign between the value of fld1 and fld2.

It is also possible to specify that the first field (in the following case fld1) should be stored as an object for the text line in the strings list.

```
mt.Extract('fld1;fld2',slData,'',true);
```

Then you can access that value by:

```
var
  v:variant;
begin
   …
   v:=TkbmVariantObject(slData.Objects[i] ).Value;
   s:=slData.Strings[i];
   …
end;
```

Its practical if the data of field fld1 is required to be quickly accessible in your string list. In this example, s will only contain the value of fld2. If you need the field value both as an object and as part of the string, you can include it twice in your field list. Eg. 'fld1;fld1;fld2'.

A different extraction method is GetRows which returns requested fields as an array of an array of variant.

```
var
  v:variant;
begin
  v:=mt.GetRows(kbmGetRowsRest, kbmBookMarkFirst,1);
end;
```

This will return an array of variant containing an array of variant of size 1, with the contents of the field with FieldNo=1.

Eg. v[0,0] is the value of the first field of the first record.

Since we specified to ask for kbmGetRowsRest number of records, starting with kbmBookMarkFirst, then we are essentially asking for all records. If we wanted to start from another place in the record set, you should provide a TBookmark value for that place. That is created by navigating to the record, then use the Bookmark function to obtain a bookmark for exactly that place in the record set.

And by providing a number instead of kbmGetRowsRest, you can limit the number of records returned to that particular count.

The last argument of GetRows, can either be a field number, a field name or an array of field numbers or an array of field names.

```
  v:=mt.GetRows(kbmGetRowsRest, kbmBookMarkFirst,VarArrayOf('fld1','fld2'));
```

Statistics

kbmMemTable contains a number of functions to allow very fast grouping and calculation of statistical values for your data.

Returning a sum of fld1 for all records visible in the current index.

```
var
   v:variant;
begin
   v:=mt.Aggregate('fld1:SUM');
end;
```

v will contain the sum of fld1 for all records in the current index.

```
var
   v:variant;
begin
   v:=mt.Aggregate('fld1:SUM;fld1:STDDEV;fld1:COUNT');
end;
```

v[0] will contain the sum of fld1 for all records in the current index. v[1] will contain the standard deviation for the records, and v[2] will contain the count of records.

The following functions can be used:

MAX, **MIN**, **AVG**, **COUNT**, SUM, STDDEV and USR1,USR2,USR3.

The USRx functions are special functions, where the developer have to provide the calculation in the eventhandler OnUserAggregate.

If you have data that you want to aggregate in groups, like how big sales in each country, then you can use the GroupBy methods.

As grouping often will result in a number of records with aggregated data, then GroupBy require a destination memory table to put the result in.

```
var
    mtGrouped:TkbmMemTable;
begin
   mtGrouped:=TkbmMemTable.Create(nil);
   try
```

```
      mt.GroupBy(mtGrouped,'fldCountry','fldCountry;fldSales:SUM;fldSales:MAX;fldSales:COUNT');
   …
   finally
      mtGrouped.Free;
   end;
end;
```

The mtGrouped table will be left open for you, with the fields named fldCountry, fldSales_SUM, fldSales_MAX and fldSales_COUNT defined in it.

Having the following values in mt

fldCountry	**fldSales**
US	10
GB	20
US	30
DE	15
DE	5
GB	10
US	12

Will result in the following values in mtGrouped

fldCountry	**fldSales_SUM**	**fldSales_MAX**	**fldSales_COUNT**
DE	20	15	2
GB	30	20	2
US	52	30	3

There are loads of more features available in kbmMemTable and this article touches only a subset of them. But go exploring and you will find attached datasets, which is multiple datasets sharing the same data without holding a copy of it.. essentially an advanced version of having multiple cursors into the data, SQL support, complex math expression evaluation via the SQL manager, locale support etc.

Go have fun ☺