How to write in-memory database applications in Lazarus/FPC

From Lazarus wiki

| **English (en)** | **français (fr)** | **русский (ru)** |

Contents

Database portal

References:

- General info
- Libraries
- Field types
- Controls
- FAQ
- SQL how-to
- Working With TSQLQuery
- In-memory database applications

Tutorials/practical articles:

- Overview
- 0 - Database set-up
- 1 - Getting started
- 2 - Editing
- 3 - Queries
- 4 - Data modules
- SQLdb Programming Reference

Databases

Advantage - MySQL - MSSQL - Postgres - Interbase - Firebird - Oracle - ODBC - Paradox - SQLite - dBASE - MS Access - Zeos

Introduction

There are certain circumstances when in-memory datasets make sense. If you need a fast, single-user, non mission-critical, non SQL database, without need for transactions, TMemDataset could suit your needs.

Some benefits are:

- Fast execution. Since all processing is done in memory, no data is saved on hard disk until explicitly asked. Memory is surely faster than hard disk.
- No need for external libraries (no .so or .dll files), no need for server installation
- Code is multiplatform and can be compiled on any OS instantly
- Since all programming is done in Lazarus/FPC, such applications are easier for maintenance. Instead of constantly switching from back-end programming to front-end programming, by using MemDatasets you can concentrate on your Pascal code.

Note: later on in this article, BufDataset is introduced. TBufDataset often is a better choice than TMemDataset

I will illustrate how to program relational non-SQL memory databases, focusing on enforcing relation integrity and filtering, simulating auto-increment primary fields and similar.

This page shares with you what I have learned experimenting with TMemDatasets. There might be some other,

more efficient way to do this. If so, please, feel free to contribute to this document for the benefit of the Lazarus/FPC community.

The memds unit provides TMemDataset, so you will need to add that to your uses clause.

Saving MemDatasets to persistent files

In the interface part of your code, declare an array type for storing information about all the TMemDataSets that you want to make persistent at the end of a session and restore at the beginning of the next session. You have to declare a variable of type TSaveTables, too.

I also use a global variable vSuppressEvents of type boolean, for suppressing Dataset events used for referential integrity enforcement, during data restore.

You get this:

```
type
  TSaveTables=array[1..15] of TMemDataset;
var
  //Global variable that holds tables for saving/restoring session
  vSaveTables:TSaveTables;
  //Suppress events flag variables. Used during data loading from files.
  vSuppressEvents:Boolean;
```

Instead of using global variables like I did, you could make them a property of the main form, also. TMemDataset has a way to natively store data to persistent file: the SaveToFile method. But, you could rather choose to save data to CSV files for easier external post processing. Therefore, I will combine both ways into same procedures. I define a constant cSaveRestore in the Interface part, by which I can define whether data will be stored and loaded as native MemDataset files or CSV files.

```
const
  //Constant cSaveRestore determines the way for saving and restoring of MemDatasets to persistent files
  cSaveRestore=0; //0=MemDataset native way, 1=saving and restoring from CSV
```

Now, you can save MemDatasets on FormClose event and load them on FormCreate event. Instantiate elements of the array of MemDatasets on the FormCreate event, too.

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  //List of tables to be saved/restored for a session
  vSaveTables[1]:=Products;
  vSaveTables[2]:=Boms;
  vSaveTables[3]:=Stocks;
  vSaveTables[4]:=Orders;
  vSaveTables[5]:=BomCalculationProducts;
  vSaveTables[6]:=BomCalculationComponents;
  vSaveTables[7]:=BomCalculationFooter;
  vSaveTables[8]:=BomCalculationProductsMultiple;
  vSaveTables[9]:=BomCalculationComponentsMultiple;
  vSaveTables[10]:=BomCalculationFooterMultiple;
  vSaveTables[11]:=ImportVariants;
  vSaveTables[12]:=ImportToTables;
  vSaveTables[13]:=ImportToFields;
  vSaveTables[14]:=ImportFromTables;
  vSaveTables[15]:=ImportFromFields;
  //Restore session
  RestoreSession;
  GetAutoincrementPrimaryFields;
end;
```

```
procedure TMainForm.FormClose(Sender: TObject; var CloseAction: TCloseAction);
begin
 //Save memdatasets to files (to save current session)
 SaveSession;
end;
```

```
procedure RestoreSession;
var
  I:Integer;
begin
  try
    MemoMessages.Append(TimeToStr(Now())+' Starting restoration of previously saved session.');
    vSuppressEvents:=True; //Supress events used for referential integrity enforcing
    //Disable controls and refresh all datasets
    for I:=Low(vSaveTables) to High(vSaveTables) do begin
      vSaveTables[I].DisableControls;
      vSaveTables[I].Refresh; //Important if dataset was filtered
    end;
    //Load memdatasets from files (to restore previous session)
    for I:=Low(vSaveTables) to High(vSaveTables) do begin
      vSaveTables[I].First;
      MemoMessages.Append(TimeToStr(Now())+' Starting restoration of table: '+vSaveTables[I].Name);
      try
        //If data is loaded from a csv file, then table must be deleted first.
        if cSaveRestore=1 then begin
          MemoMessages.Append(TimeToStr(Now())+' Starting delete of all records in table:
'+vSaveTables[I].Name);
          //This way of deleting all records is incredibly slow.
          {while not vSaveTables[I].EOF do begin
            vSaveTables[I].Delete;
          end;}
          //This method for deleting of all records is much faster
          EmptyMemDataSet(vSaveTables[I]);
          MemoMessages.Append(TimeToStr(Now())+' All records from table: '+vSaveTables[I].Name+' deleted.');
        end;
      except
        on E:Exception do begin
          MemoMessages.Append(TimeToStr(Now())+' Error while deleteing records from table: '+vSaveTables[I].Name
+'. '+E.Message);
        end;
      end;
      try
        try
          MemoMessages.Append(TimeToStr(Now())+' Restoring table: '+vSaveTables[I].Name);
          //Check constant for way of saving/restoring data and load saved session
          case cSaveRestore of
            0:vSaveTables[I].LoadFromFile(vSaveTables[I].Name);
            1:LoadFromCsv(vSaveTables[I]);
          end;
        except
          on E:Exception do begin
            MemoMessages.Append(TimeToStr(Now())+' Error while restoring table: '+vSaveTables[I].Name +'.
'+E.Message);
          end;
        end;
      finally
        vSaveTables[I].Active:=True;//Needed because of LoadFromFile method....
      end;
      MemoMessages.Append(TimeToStr(Now())+' Table: '+vSaveTables[I].Name+' restored.');
    end;
  finally
    vSuppressEvents:=False;
    //Refresh all datasets  and enable controls
    for I:=Low(vSaveTables) to High(vSaveTables) do begin
      vSaveTables[I].Refresh; //Needed for tables that are filtered.
      vSaveTables[I].EnableControls;
    end;
     MemoMessages.Append(TimeToStr(Now())+' All tables restored from saved files.');
  end;
end;
```

```
procedure SaveSession;
var
  I:Integer;
```

```
begin
  try
    MemoMessages.Append(TimeToStr(Now())+' Starting saving session to persistent files.');
    vSuppressEvents:=True;
    //Disable controls and refresh all datasets
    for I:=Low(vSaveTables) to High(vSaveTables) do begin
      vSaveTables[I].DisableControls;
      vSaveTables[I].Refresh; //Important if dataset was filtered
    end;
    //Save session to file
    for I:=Low(vSaveTables) to High(vSaveTables) do begin
      vSaveTables[I].First;
      MemoMessages.Append(TimeToStr(Now())+' Saving table: '+vSaveTables[I].Name);
      try
        //Check constant for way of saving/restoring data and save session
        case cSaveRestore of
          0:vSaveTables[I].SaveToFile(vSaveTables[I].Name);
          1:SaveToCsv(vSaveTables[I]);
        end;
      except
        on E:Exception do begin
          MemoMessages.Append(TimeToStr(Now())+' Error while saving table: '+vSaveTables[I].Name +'.
'+E.Message);
        end;
      end;
      MemoMessages.Append(TimeToStr(Now())+' Table: '+vSaveTables[I].Name+' saved.');
    end;
  finally
    vSuppressEvents:=False;
    //Refresh all datasets  and enable controls
    for I:=Low(vSaveTables) to High(vSaveTables) do begin
      vSaveTables[I].Refresh; //Needed for tables that are filtered
      vSaveTables[I].EnableControls;
    end;
     MemoMessages.Append(TimeToStr(Now())+' All tables saved to files.');
  end;
end;
```

```
procedure EmptyMemDataSet(DataSet:TMemDataSet);
var
  vTemporaryMemDataSet:TMemDataSet;
  vFieldDef:TFieldDef;
  I:Integer;
begin
  try
    //Create temporary MemDataSet
    vTemporaryMemDataSet:=TMemDataSet.Create(nil);
    //Store FieldDefs to Temporary MemDataSet
    for I:=0 to DataSet.FieldDefs.Count-1 do begin
      vFieldDef:=vTemporaryMemDataSet.FieldDefs.AddFieldDef;
      with DataSet.FieldDefs[I] do begin
        vFieldDef.Name:=Name;
        vFieldDef.DataType:=DataType;
        vFieldDef.Size:=Size;
        vFieldDef.Required:=Required;
      end;
    end;
    //Clear existing fielddefs
    DataSet.Clear;
    //Restore fielddefs
    DataSet.FieldDefs:=vTemporaryMemDataSet.FieldDefs;
    DataSet.Active:=True;
  finally
  vTemporaryMemDataSet.Clear;
  vTemporaryMemDataSet.Free;
  end;
end;
```

```
procedure LoadFromCsv(DataSet:TDataSet);
var
  vFieldCount:Integer;
  I:Integer;
begin
  try
    //Assign SdfDataSetTemporary
```

```
    with SdfDataSetTemporary do begin
      Active:=False;
      ClearFields;
      FileName:=DataSet.Name+'.txt';
      FirstLineAsSchema:=True;
      Active:=True;
      //Determine number of fields
      vFieldCount:=FieldDefs.Count;
    end;
    //Iterate through SdfDataSetTemporary and insert records into MemDataSet
    SdfDataSetTemporary.First;
    while not SdfDataSetTemporary.EOF do begin
      DataSet.Append;
      //Iterate through FieldDefs
      for I:=0 to vFieldCount-1 do begin
        try
          DataSet.Fields[I].Value:=SdfDataSetTemporary.Fields[I].Value;
        except
          on E:Exception do begin
            MemoMessages.Append(TimeToStr(Now())+' Error while setting value for field: '
             +DataSet.Name+'.'+DataSet.Fields[I].Name +'. '+E.Message);
          end;
        end;
      end;
      try
        DataSet.Post;
      except
        on E:Exception do begin
          MemoMessages.Append(TimeToStr(Now())+' Error while posting record to table: '
           +DataSet.Name+'.'+E.Message);
        end;
      end;
      SdfDataSetTemporary.Next;
    end;
  finally
    SdfDataSetTemporary.Active:=False;
    SdfDataSetTemporary.ClearFields;
  end;
end;
```

```
procedure SaveToCsv(DataSet:TDataSet);
var
  myFileName:string;
  myTextFile: TextFile;
  i: integer;
  s: string;
begin
  myFileName:=DataSet.Name+'.txt';
  //create a new file
  AssignFile(myTextFile, myFileName);
  Rewrite(myTextFile);
  s := ''; //initialize empty string
  try
    //write field names (as column headers)
    for i := 0 to DataSet.Fields.Count - 1 do
      begin
        s := s + Format('%s,', [DataSet.Fields[i].FieldName]);
      end;
    Writeln(myTextFile, s);
    DataSet.First;
    //write field values
    while not DataSet.Eof do
      begin
        s := '';
        for i := 0 to DataSet.FieldCount - 1 do
          begin
            //Numerical fields without quotes, string fields with quotes
            if ((DataSet.FieldDefs[i].DataType=ftInteger)
             or (DataSet.FieldDefs[i].DataType=ftFloat)) then
              s := s + Format('%s,', [DataSet.Fields[i].AsString])
            else
              s := s + Format('"%s",', [DataSet.Fields[i].AsString]);
          end;
        Writeln(myTextfile, s);
        DataSet.Next;
      end;
  finally
```

```
    CloseFile(myTextFile);
  end;
end;
```

Autoincrement Primary Keys

Autoincrement field type is not supported by MemDataset. Nevertheless, you can imitate it by using Integer field type and providing a calculator for autoincrement fields. We need global variables or public properties for storing current autoincrement field value. I prefer global variables, declared in Interface part.

```
var
  //Global variables used for calculation of autoincrement primary key fields of MemDatasets
  vCurrentId:Integer=0;
  vProductsId:Integer=0;
  vBomsId:Integer=0;
  vBomCalculationProductsId:Integer=0;
  vBomCalculationComponentsId:Integer=0;
  vBomCalculationFooterId:Integer=0;
  vBomCalculationProductsMultipleId:Integer=0;
  vBomCalculationComponentsMultipleId:Integer=0;
  vBomCalculationFooterMultipleId:Integer=0;
  vStocksId:Integer=0;
  vOrdersId:Integer=0;
  vImportVariantsId:Integer=0;
  vImportToTablesId:Integer=0;
  vImportToFieldsId:Integer=0;
  vImportFromTablesId:Integer=0;
  vImportFromFieldsId:Integer=0;
```

Then we have a procedure for autoincrement field values calculation:

```
procedure GetAutoincrementPrimaryFields;
var
  I:Integer;
  vId:^Integer;
begin
  try
    MemoMessages.Lines.Append(TimeToStr(Now())+' Getting information about autoincrement fields');
    vSuppressEvents:=True;
    //Disable controls and refresh all datasets
    for I:=Low(vSaveTables) to High(vSaveTables) do begin
      vSaveTables[I].DisableControls;
      vSaveTables[I].Refresh; //Important if dataset was filtered
    end;
    for I:=Low(vSaveTables) to High(vSaveTables) do begin
      with vSaveTables[I] do begin
        //Use appropriate global variable
        case StringToCaseSelect(Name,
          ['Products','Boms','Stocks','Orders',
            'BomCalculationProducts','BomCalculationComponents','BomCalculationFooter',
            'BomCalculationProductsMultiple','BomCalculationComponentsMultiple','BomCalculationFooterMultiple',
            'ImportVariants','ImportToTables','ImportToFields','ImportFromTables','ImportFromFields']) of
          0:vId:=@vProductsId;
          1:vId:=@vBomsId;
          2:vId:=@vStocksId;
          3:vId:=@vOrdersId;
          4:vId:=@vBomCalculationProductsId;
          5:vId:=@vBomCalculationComponentsId;
          6:vId:=@vBomCalculationFooterId;
          7:vId:=@vBomCalculationProductsMultipleId;
          8:vId:=@vBomCalculationComponentsMultipleId;
          9:vId:=@vBomCalculationFooterMultipleId;
          10:vId:=@vImportVariantsId;
          11:vId:=@vImportToTablesId;
          12:vId:=@vImportToFieldsId;
          13:vId:=@vImportFromTablesId;
          14:vId:=@vImportFromFieldsId;
        end;
```

```
        try
          //Find last value of Id and save it to global variable
          Last;
          vCurrentId:=FieldByName(Name+'Id').AsInteger;
          if (vCurrentId>vId^) then vId^:=vCurrentId;
        finally
          //Remove reference;
          vId:=nil;
        end;
      end;
    end;
  finally
    vSuppressEvents:=False;
    //Refresh all datasets  and enable controls
    for I:=Low(vSaveTables) to High(vSaveTables) do begin
      vSaveTables[I].Refresh;
      vSaveTables[I].EnableControls;
    end;
     MemoMessages.Lines.Append(TimeToStr(Now())+' Autoincrement fields - done.');
  end;
end;
```

```
function StringToCaseSelect(Selector:string;CaseList:array of string):Integer;
var
  cnt: integer;
begin
  Result:=-1;
  for cnt:=0 to Length(CaseList)-1 do
  begin
    if CompareText(Selector, CaseList[cnt]) = 0 then
    begin
      Result:=cnt;
      Break;
    end;
  end;
end;
```

The GetAutoincrementPrimaryFields procedure is called every time after you restore (load) data from persistent files, in order to load last autoincrement values into global variables (or properties, as you prefer). Autoincrementing is done in OnNewRecord event of every MemDataset. For example, for MemDataset Orders:

```
procedure TMainForm.OrdersNewRecord(DataSet: TDataSet);
begin
  if vSuppressEvents=True then Exit;
  //Set new autoincrement value
  vOrdersId:=vOrdersId+1;
  DataSet.FieldByName('OrdersId').AsInteger:=vOrdersId;
end;
```

As already explained, I use vSuppressEvents global variable as flag for the case of restoring data from persistent files.

Enforcing Referential Integrity

There is no enforced referential integrity implemented in MemDataset component, so you have to do it on your own.

Let's assume we have two tables: MasterTable and DetailTable.

There are various places where referential integrity code needs to be used:

- Insert/Update code is located in the `BeforePost` event of the DetailTable: before a new/updated detail record is posted/saved, it needs to be checked for meeting referential integrity requirements

- Delete code is located in the `BeforeDelete` event of the MasterTable: before a master record is deleted, it needs to make sure any child records meet referential integrity requirements

```
procedure TMainForm.MasterTableBeforeDelete(DataSet: TDataSet);
begin
  if vSuppressEvents=True then Exit;
  try
    DetailTable.DisableControls;
    // Enforce referential delete ("cascade delete") for table "MasterTable"
    while not DetailTable.EOF do begin
      DetailTable.Delete;
    end;
    DetailTable.Refresh;
  finally
    DetailTable.EnableControls;
  end;
end;
```

```
procedure TMainForm.DetailTableBeforePost(DataSet: TDataSet);
begin
  if vSuppressEvents=True then Exit;
  // Enforce referential insert/update for table "DetailTable" with
  // foreign key "MasterTableID" linking to
  // the MasterTable ID primary key field
  DataSet.FieldByName('MasterTableId').AsInteger:=
    MasterTable.FieldByName('ID').AsInteger;
end;
```

After you provided referential Insert/Update/Delete, all you must do is provide code for master/detail filtering of data. You do it in the `AfterScroll` event of the MasterTable and in the `OnFilter` event of the DetailTable.

Don't forget to set the `Filtered` property of DetailTable to True.

```
procedure TMainForm.MasterTableAfterScroll(DataSet: TDataSet);
begin
  if vSuppressEvents=True then Exit;
  DetailTable.Refresh;
end;
```

```
procedure TMainForm.DetailTableFilterRecord(DataSet: TDataSet;
  var Accept: Boolean);
begin
  if vSuppressEvents=True then Exit;
  // Show only child fields whose foreign key points to current
  // master table record
  Accept:=DataSet.FieldByName('MasterTableId').AsInteger=
    MasterTable.FieldByName('ID').AsInteger;
end;
```

Known problems

There are several limitations when using MemDatasets.

- Locate method does not work
- Filtering by using Filter and Filtered property does not work. You must use hardcoding in the OnFilter event.
- Looping deletion of records seems to be incredibly slow. Therefore I use my EmptyMemDataset procedure instead of while not EOF do Delete;
- In FPC 2.6.x and earlier, CopyFromDataSet method copies data only from the current cursor position to the end of the source dataset. So, you have to write MemDataset1.First; before

MemDataSet2.CopyFromDataSet(MemDataset1);. Fixed in FPC trunk revision 26233.

- Note that older versions of FPC has no CopyFromDataset in Bufdataset, at the time an advantage for MemDs.
- See bug report http://bugs.freepascal.org/view.php?id=25426.

TBufDataSet

As previously mentioned, MemDataSet lacks custom filters, autoincrement data type and the Locate method, so it is better to use TBufDataSet instead. TBufDataset is provided by the BufDataset unit.

Since there is no component for design-time editing of TBufDataSet (but you can set up field definitions at design time), you could create a custom wrapper component or use it through code, in the same way as ClientDataSet in Delphi. Look at the Delphi documentation relating to client datasets for details.

You can use the same methods for enforcing referential integrity and primary autoincrement fields as explained for MemDataSet.

There are only small differences between MemDataSet and BufDataset:

MemDataSet	BufDataset
DataSet.ClearFields	DataSet.Fields.Clear
DataSet.CreateTable	DataSet.CreateDataSet

Sorting DBGrid on TitleClick event for TBufDataSet

If you wish to enable consecutive ascending and descending sorting of a DBGrid showing some data from TBufDataSet, you could use the following method:

```
Uses
  BufDataset, typinfo;

function SortBufDataSet(DataSet: TBufDataSet;const FieldName: String): Boolean;
var
  i: Integer;
  IndexDefs: TIndexDefs;
  IndexName: String;
  IndexOptions: TIndexOptions;
  Field: TField;
begin
  Result := False;
  Field := DataSet.Fields.FindField(FieldName);
  //If invalid field name, exit.
  if Field = nil then Exit;
  //if invalid field type, exit.
  if {(Field is TObjectField) or} (Field is TBlobField) or
    {(Field is TAggregateField) or} (Field is TVariantField)
     or (Field is TBinaryField) then Exit;
  //Get IndexDefs and IndexName using RTTI
  if IsPublishedProp(DataSet, 'IndexDefs') then
    IndexDefs := GetObjectProp(DataSet, 'IndexDefs') as TIndexDefs
  else
    Exit;
  if IsPublishedProp(DataSet, 'IndexName') then
    IndexName := GetStrProp(DataSet, 'IndexName')
  else
    Exit;
  //Ensure IndexDefs is up-to-date
  IndexDefs.Updated:=false; {<<<<---This line is critical as IndexDefs.Update will do nothing on the next sort
```

```
if it's already true}
  IndexDefs.Update;
  //If an ascending index is already in use,
  //switch to a descending index
  if IndexName = FieldName + '__IdxA'
  then
    begin
      IndexName := FieldName + '__IdxD';
      IndexOptions := [ixDescending];
    end
  else
    begin
      IndexName := FieldName + '__IdxA';
      IndexOptions := [];
    end;
  //Look for existing index
  for i := 0 to Pred(IndexDefs.Count) do
  begin
    if IndexDefs[i].Name = IndexName then
      begin
        Result := True;
        Break
      end;  //if
  end; // for
  //If existing index not found, create one
  if not Result then
      begin
        if IndexName=FieldName + '__IdxD' then
          DataSet.AddIndex(IndexName, FieldName, IndexOptions, FieldName)
        else
          DataSet.AddIndex(IndexName, FieldName, IndexOptions);
        Result := True;
      end; // if not
  //Set the index
  SetStrProp(DataSet, 'IndexName', IndexName);
end;
```

So, you can call this function from a DBGrid in this way:

```
procedure TFormMain.DBGridProductsTitleClick(Column: TColumn);
begin
  SortBufDataSet(Products, Column.FieldName);
end;
```

Sorting multiple columns in grid

I have written TDBGridHelper for sorting grid by multiple columns while holding shift key. Note MaxIndexesCount must be set quite large for TBufDataSet because there can be quite large combinations of possible sorting options. But I think people would not use more than 10 so setting it 100 should be teoretically Ok.

```
  { TDBGridHelper }

  TDBGridHelper = class helper for TDBGrid
  public const
    cMaxColCOunt = 3;
  private
    procedure Interbal_MakeNames(Fields: TStrings; out FieldsList, DescFields: String);
    procedure Internal_SetColumnsIcons(Fields: TStrings; AscIdx, DescIdx: Integer);
    function Internal_IndexNameExists(IndexDefs: TIndexDefs; IndexName: String): Boolean;
  public
    procedure Sort(const FieldName: String; AscIdx: Integer = -1; DescIdx: Integer = -1);
    procedure ClearSort;
  end;
```

```
{ TDBGridHelper }

procedure TDBGridHelper.Interbal_MakeNames(Fields: TStrings; out FieldsList, DescFields: String);
var
  FldList: TStringList;
  DscList: TStringList;
  FldDesc, FldName: String;
  i: Integer;
begin
  if Fields.Count = 0 then
  begin
    FieldsList := '';
    DescFields := '';
    Exit;
  end;

  FldList := TStringList.Create;
  DscList := TStringList.Create;
  try
    FldList.Delimiter := ';';
    DscList.Delimiter := ';';

    for i := 0 to Fields.Count - 1 do
    begin
      Fields.GetNameValue(i, FldName, FldDesc);
      FldList.Add(FldName);

      if FldDesc = 'D' then
        DscList.Add(FldName);
    end;

    FieldsList := FldList.DelimitedText;
    DescFields := DscList.DelimitedText;
  finally
    FldList.Free;
    DscList.Free;
  end;
end;

procedure TDBGridHelper.Internal_SetColumnsIcons(Fields: TStrings; AscIdx, DescIdx: Integer);
var
  i: Integer;
  FldDesc: String;
begin
  for i := 0 to Self.Columns.Count - 1 do
  begin
    FldDesc := Fields.Values[Self.Columns[i].Field.FieldName];

    if FldDesc = 'A' then
      Self.Columns[i].Title.ImageIndex := AscIdx
    else
    if FldDesc = 'D' then
      Self.Columns[i].Title.ImageIndex := DescIdx
    else
      Self.Columns[i].Title.ImageIndex := -1
  end;
end;

function TDBGridHelper.Internal_IndexNameExists(IndexDefs: TIndexDefs; IndexName: String): Boolean;
var
  i: Integer;
begin
  for i := 0 to IndexDefs.Count - 1 do
  begin
    if IndexDefs[i].Name = IndexName then
      Exit(True)
  end;

  Result := False
end;

procedure TDBGridHelper.Sort(const FieldName: String; AscIdx: Integer;
  DescIdx: Integer);
var
  Field: TField;
  DataSet: TBufDataset;
  IndexDefs: TIndexDefs;
  IndexName, Dir, DescFields, FieldsList: String;
  Fields: TStringList;
```

```
begin
  if not Assigned(DataSource.DataSet) or
     not DataSource.DataSet.Active or
     not (DataSource.DataSet is TBufDataset) then
    Exit;
  DataSet := DataSource.DataSet as TBufDataset;

  Field := DataSet.FieldByName(FieldName);
  if (Field is TBlobField) or (Field is TVariantField) or (Field is TBinaryField) then
    Exit;

  IndexDefs := DataSet.IndexDefs;
  IndexName := DataSet.IndexName;

  if not IndexDefs.Updated then
    IndexDefs.Update;

  Fields := TStringList.Create;
  try
    Fields.DelimitedText := IndexName;
    Dir := Fields.Values[FieldName];

    if Dir = 'A' then
      Dir := 'D'
    else
    if Dir = 'D' then
      Dir := 'A'
    else
      Dir := 'A';

    //If shift is presed then add field to field list
    if ssShift in GetKeyShiftState then
    begin
      Fields.Values[FieldName] := Dir;
      //We do not add to sor any more field if total field count exids cMaxColCOunt
      if Fields.Count > cMaxColCOunt then
        Exit;
    end
    else
    begin
      Fields.Clear;
      Fields.Values[FieldName] := Dir;
    end;

    IndexName := Fields.DelimitedText;
    if not Internal_IndexNameExists(IndexDefs, IndexName) then
    begin
      Interbal_MakeNames(Fields, FieldsList, DescFields);
      TBufDataset(DataSet).AddIndex(IndexName, FieldsList, [], DescFields, '');
    end;

    DataSet.IndexName := IndexName;
    Internal_SetColumnsIcons(Fields, AscIdx, DescIdx)
  finally
    Fields.Free;
  end;
end;

procedure TDBGridHelper.ClearSort;
var
  DataSet: TBufDataset;
  Fields: TStringList;
begin
  if not Assigned(DataSource.DataSet) or
     not DataSource.DataSet.Active or
     not (DataSource.DataSet is TBufDataset) then
    Exit;
  DataSet := DataSource.DataSet as TBufDataset;

  DataSet.IndexName := '';

  Fields := TStringList.Create;
  try
    Internal_SetColumnsIcons(Fields, -1, -1)
  finally
    Fields.Free
  end
end;
```

To use sorting you need to call helper methods in OnCellClick and onTitleClick. OnTitleClick - If you hold shift ads new column to sot list ore changes direction to selected column or just sorts one column OnCellClick - If you double click on cell[0, 0] grid clears its sorting

```
procedure TForm1.grdCountriesCellClick(Column: TColumn);
begin
  if not Assigned(Column) then
    grdCountries.ClearSort
end;

procedure TForm1.grdCountriesTitleClick(Column: TColumn);
begin
  grdCountries.Sort(Column.Field.FieldName, 0, 1);
end;
```

If you have assigned TitleImageList then you can specify which image use for ascending and which for descending operations.

ZMSQL

Another, often better way to write in-memory databases is to use the ZMSQL package:

- ZMSQL
- http://sourceforge.net/projects/lazarus-ccr/files/zmsql/ (http://sourceforge.net/projects/lazarus-ccr/files/zmsql/)
- http://www.lazarus.freepascal.org/index.php/topic,13821.30.html (http://www.lazarus.freepascal.org/index.php/topic,13821.30.html)

Contributors

Original text written by: Zlatko Matić (matalab@gmail.com)

Other contributions by contributors as shown in the page History.

Retrieved from "https://wiki.lazarus.freepascal.org/index.php?title=How_to_write_in-memory_database_applications_in_Lazarus/FPC&oldid=116502"
