janSQL

janSQL

TjanSQL is a single user relational Database engine implemented as a Delphi object using plain text files with semi-colon seperated data for data storage. Supported SQL: SELECT (with table joins, field aliases and calculated), UPDATE, INSERT (values and sub-select), DELETE, CREATE TABLE, DROP TABLE, ALTER TABLE, CONNECT TO, COMMIT, WHERE (rich bracketed expression), IN (list or sub query), GROUP BY, HAVING, ORDER BY (ASC, DESC), nested sub queries, statistics (COUNT, SUM, AVG, MAX, MIN), operators (+,-,*,/, and, or,>,>=,<,<=,=,<>,Like), functions (UPPER, LOWER, TRIM, LEFT, MID, RIGHT, LEN, FIX, SOUNDEX, SQR, SQRT). High performance: complete in-memory handling of tables and recordsets; semi-compiled expressions. Released under MOZILLA PUBLIC LICENSE Version 1.1. NEW FEATURES: fixed memory leak, calculated fields (in select and update statements), field aliases, table aliases, join "unlimited" tables, stdDev aggregate function, ASSIGN TO for named temporary tables, SAVE TABLE for persisting recordsets, INSERT INTO, ISO 8601 dates, numerous extra functions.

Intended use

janSQL is intended for single-user desktop application where you want to access and update data using SQL without having to deploy the Borland Database engine or the Microsoft Data Access Components. janSQL is not intended to be used with the Delphi data access components. The included demo shows that it is very easy to create a user interface for janSQL, displaying data in a TStringGrid.

Motivation

When developing my PascalServer (a virtual web server that can serve Pascal Script pages to Dave Baldwin's THTMLViewer) I needed a fast and simple to use database engine that could be compiled into PascalServer. Althoug there are some good freeware Delphi database components (both for in-memory tables and for handling dbf files), I could not find a freeware component that would allow me to work with SQL (including joining of tables). So I started writing my own SQL DBMS.

License

janSQL is released under the MOZILLA PUBLIC LICENSE Version 1.1.

janSQK makes use of the mwStringHashList component created by Martin Martin Waldenburg (Martin.Waldenburg@T-Online.de). The source code of mwStringHashList is included. mwStringHashList was also released under MPL version 1.1.

This means that you are allowed to include janSQL in your freeware and commercial projects as long as you distribute the janSQL source code with your product and as long as you leave the MPL statements in the body of the source code intact and clearly indicate any modifications that you made.

New

New

janSQL was originally released on 24-Mar-2002 as version 1.0.

version 1.1 of 25-Mar-2002

- closed memory leaks
- allow "unlimited" number of tables in a join
- allow calculated updates: set field=expression
- table aliases
- StdDev aggregate function
- RecordFields are true objects

Updates

Updates

janSQL is written by Jan Verhoeven with Delphi 5.

Email

jan1.verhoeven@wxs.nl

WebSite

http://jansfreeware.com

Get started

Get started

If you are new to SQL (Structured Query Language) then you should first learn about SQL (either by reading a good book about it or by visiting an on-line tutorial e.g. http://www.w3schools.com).

janSQLDemo

Run this simple demo program that allows you to execute SQL statements and see the result of SELECT statements.

Connect

Before you can access data you must connect to a database. In janSQL a database is a folder. Using the CONNECT TO statement you can connect to a database. Just click the Execute button to execute the statement(s) in the SQL editor. When everything goes right you will see OK in the message box next to the Execute button.

Commit

Nothing is saved to disk untill you issue the COMMIT statement. Not only does this make janSQL very fast but it also allows you to play with it without altering any disk based data.

Select

Enter the following simple select statement in the editor and click Execute:

```
  select * from users
```

In the table grid you will see all records from the users table, with the field names displayed in the header of the grid.

More SQL

For other SQL instruction see the SQL Syntax chapter where all supported SQL statements are explained.

Multiple SQL statements

You can enter multiple, semi-colon ; seperated, SQL statements in the SQL editor. Upon clicking Execute each statement will be processed on turn.

Retrieving data

Retrieving data

When you use janSQL in your own programs you obviously want to retrieve the data from a resulting recordset. Below you see the complete source code of janSQLDemo.

```
unit janSQLDemoU;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  FileCtrl,Grids, ExtCtrls, ComCtrls, ToolWin, Menus, janSQL, StdCtrls, Buttons;

type
  TjanSQLDemoF = class(TForm)
    MainMenu1: TMainMenu;
    ToolBar1: TToolBar;
    StatusBar1: TStatusBar;
    Panel1: TPanel;
    Splitter1: TSplitter;
    viewgrid: TStringGrid;
    sqlmemo: TMemo;
    cmdExecute: TSpeedButton;
    edmessage: TEdit;
    Insert1: TMenuItem;
    ApplicationFolder1: TMenuItem;
    SelectedFolder1: TMenuItem;
    Help1: TMenuItem;
    Contents1: TMenuItem;
    procedure FormCreate(Sender: TObject);
    procedure cmdExecuteClick(Sender: TObject);
    procedure ApplicationFolder1Click(Sender: TObject);
    procedure SelectedFolder1Click(Sender: TObject);
    procedure Contents1Click(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
  private
    procedure showresults(resultset:integer);
     Private declarations
  public
     Public declarations
  end;

var
  janSQLDemoF: TjanSQLDemoF;
  appldir:string;
  db:TjanSQL;

implementation

{$R *.DFM}

procedure TjanSQLDemoF.FormCreate(Sender: TObject);
begin
  appldir:=extractfiledir(application.exename);
  db:=TjanSQL.create;
  sqlmemo.Text:='connect to '''+appldir+'''';
end;
```

```
procedure TjanSQLDemoF.cmdExecuteClick(Sender: TObject);
var
  sqlresult:integer;
  sqltext:string;
begin
  sqltext:=sqlmemo.text;
  sqlresult:=db.SQLDirect(sqltext);
  if sqlresult<>0 then begin
    edmessage.Text:='OK';
    sqlmemo.text:='';
    if sqlresult>0 then begin
      showresults(sqlresult);
      db.ReleaseRecordset(sqlresult);
    end;
  end
  else
    edmessage.Text:=db.Error;
  sqlmemo.SetFocus;
end;

procedure TjanSQLDemoF.showresults(resultset:integer);
var
  r1:integer;
  i,arow,acol,c,rc,fc:integer;
begin
  r1:=resultset;
  rc:=db.RecordSets[r1].recordcount;
  if rc=0 then exit;
  fc:=db.RecordSets[r1].fieldcount;
  if fc=0 then exit;
  viewgrid.RowCount:=rc+1;
  viewgrid.ColCount:=fc;
  for i:=0 to fc-1 do
    viewgrid.Cells[i,0]:=db.recordsets[r1].FieldNames[i];
  for arow:=0 to rc-1 do
    for acol:=0 to fc-1 do
      viewgrid.cells[acol,arow+1]:=db.RecordSets[r1].records[arow].fields[acol];
end;


procedure TjanSQLDemoF.ApplicationFolder1Click(Sender: TObject);
begin
  sqlmemo.SelText:=appldir;
end;

procedure TjanSQLDemoF.SelectedFolder1Click(Sender: TObject);
var
  adir:string;
begin
  if not selectdirectory('Select Catalog Folder to insert','',adir) then exit;
  sqlmemo.SelText:=adir;
end;



procedure TjanSQLDemoF.Contents1Click(Sender: TObject);
begin
  application.HelpFile:=appldir+'.hlp';
  application.HelpJump('janSQL');
end;

procedure TjanSQLDemoF.FormDestroy(Sender: TObject);
begin
```

```
  db.free;
end;

end.
```

We use the SQLDirect method of a TjanSQL instance (in this case db) and retrieve an sqlresult. When SQLDirect returns 0 this means an error has occured. We retrieve the error string with db.Error and display it in the message box. When the returned value is >0 then a resulting recordset is returned. The return value is the (1-based) index of the recordset. Next we use showresults(sqlresult)   to display the results in a TStringGrid.

This is really all the code you need. Instead of putting the values in a TstringGrid you could also put them in an array for further manipulation, or insert them in a html template for display in a web browser.

Performance

Performance
You will notice that TjanSQL is fast because all processing is done in-memory. Although not intented for use with huge tables, 1000 records are no problem.

As memory is becoming cheaper and computers becoming faster, you will be able to process your data even better and faster in the future.

Data Formats

Data Formats
janSQL uses the point as decimal separator:

123.45 is a valid number. 123,45 is not.

Extending janSQL

Extending janSQL

Within the conditions of the MOZILLA PUBLIC LICENSE Version 1.1. you are allowed to modify and extend janSQL.

Adding new functions

Proceed as follows to add a new function to janSQL.

Suppose we want to add the Ceil function:

Ceil rounds variables up toward positive infinity.

E.g.:

```
Ceil(-2.8) = -2

Ceil(2.8) = 3

Ceil(-1.0) = -1
```

- in janSQLTokenizer: add toCeil to TTokenOperator
- in janSQLTokenizer: add ceil to the IsFunction function

```
else if value='ceil' then begin
  FtokenKind:=tkOperator;
  FTokenOperator:=toCeil;
  FtokenLevel:=7;
  result:=true;
end
```

- in janSQLExpression2: add private procedure procCeil to TjanSQLExpression2

```
procedure TjanSQLExpression2.procCeil;
var
  v1:variant;
begin
  v1:=runpop;
  runpush(ceil(v1));
end;
```

- in janSQLExpresion2: add case toCeil to the runoperator procedure:

```
 toCeil:procCeil;
```

That is all.

Data Exchange

Data Exchange

When you have tables in other database formats, e.g. Access 2000 you can export them in delimited text format.

Export from Access 2000

The included programs.csv file was exported from an (out of date) Access 2000 database:

- Open the Access database
- Select the table to export
- Select File Export - Save as Type=text
- Select Save as delimited - Next
- Select Semi-colon delimiter; include field names on first row; text qualifier: {none}.
- Enter filename when prompted and save with .txt extension

File Format

File Format
janSQL works with plain text files that have the .txt extension (to allow for quick opening in e.g. Notepad) and where data is separated by semi-colons. The first row of the file contains the field names.

The file format was choosen for easy export from Microsoft Access

No quotes are used around text fields. You can not use the semi-colon character in data fields as it is used as the field seperator.

Handling dates

Handling dates

Unlike most other database engines that have strong data typing, there are no data types in janSQL. Everything is stored as a string and in calculations converted ad-hoc to a number when applicable in the context.

To handle dates you must store them as a string in the ISO 8601 format:

```
   YYYY-MM-DD
   YYYY-MM-DDThh:mm:ss
```

A 4-digit year, followed by the 2-digit month number followed by the 2-digit day number. In your WHERE clauses you can then compare these string dates with each other. E.g. 1953-11-16 will be less than 2002-03-26.

```
  SELECT * FROM users WHERE birthday>'1980-01-01'
```

Also the ORDER BY clause will produce correct results.

Handling times

In janSQL you store dates and times in seperate fields. This is a good idea for any DMBS that you use. It makes many queries so much easier and clearer.

In janSQL you store times as a string in the hh:mm:ss format:
A 2 digit hour followed by a 2 digit minute value, followed by a 2-digit second value. Times are from '00:00:00' to '23:59:59'

Demo Introduction

Demo Introduction

janSQLDemo was written to demonstrate the use of janSQL. It is a very simple program that allows you to enter and execute SQL statements, provides any error feedback, and will show the result of SELECT queries in a stringgrid.

Connect to the database

The very first statement you must execute is the CONNECT TO statement. Just press the F9 key when you have started janSQLDemo.

Experiment

Unless you issue the COMMIT statement, all processing is done in-memory. This makes janSQL and the demo in particular, ideal for experimenting with SQL. An exception are the CREATE TABLE and the DROP TABLE statement, which are executed immediately.

Examples

janSQLDemo comes with a series of testing examples stored in the samples.txt file. You can add your own samples following the same format as the current samples. The sample title should be placed on a seperate line between square brackets.

```
[order by]
SELECT *
FROM users
ORDER BY #userid ASC, productid DESC
```

Sample titles will be automatically added to the Samples menu.

Hints

The result of a SELECT statement is displayed in the stringgrid. You will notice that when you move over a cell, the complete text of the cell is displayed as a hint. You can use the technique that I used in your own programs.

Loading and Saving

janSQL allows you to execute a batch of SQL statements seperated by a semicolon. Using the Load and Save menu options you can load and save .sql files that contain multiple SQL statements.

Component introduction

Component introduction

TjanSQL is a delphi freeware component with source code released under the MOZILLA PUBLIC LICENSE Version 1.1. In fact TjanSQL is not a component but derived from TObject.

Installation

Installation

Please the supplied source files in any folder (e.g. components\jansoft\janSQL) and include the folder in the Delphi library path.

- janSQL.pas
- janSQLExpression2.pas
- janSQLTokenizer.pas
- janSQLStrings.pas
- mwStringHashList.pas

Then you create the component as described in TjanSQL creation

janSQLStrings

janSQLStrings
A small unit with usefull string routines.

TjanSQL overview

TjanSQL overview

TjanSQL was designed for ease of use. With just a few methods you can manage a database and display selected data:

```
function SQLDirect(value:string):integer;
function ReleaseRecordset(arecordset:integer):boolean;
function Error:string;
property RecordSets[index:integer]:TjanRecordSet read getrecordset;
  property RecordSetCount:integer read getRecordSetCount;
```

SQLDirect

Will process a value that consists of one or more semi-colon ; seperated SQL statements and returns 0 in case of failure, -1 in case of valid execution with no resultset, and the number of the result recordset in case of a SELECT statement.

Recordsets

Read-only property to retrieve a reference to a recordset. In janSQL all collection (records, fields) are 0 based, except for the recordsets collection which is 1-based. In case of a select statement SQLDirect will return the index of the generated result set.

See Retrieving data for example code.

ReleaseRecordset

Allows you to release a recordset given the 1-based index of the recordset. You will normally use this after a SELECT resultset has been processed. Returns false in case of failure and true when the recordset was deleted.

RecordSetCount

Returns the number of recordsets. Can be used together with the RecordSets property and the ReleaseRecordset method to clear all recordsets.

Notes

TjanSQL and all helper components clean-up any resources they use automatically when freed.

TjanSQL creation

TjanSQL creation

You do not place TjanSQL on the component palette but create it in e.g. the FormCreate event of your form, and free it in the FormDestroy event of the form.   Include janSQL in the uses clause of the interface part of the form.

```
var
  janSQLDemoF: TjanSQLDemoF;
  appldir:string;
  thefile:string;
  db:TjanSQL;


procedure TjanSQLDemoF.FormCreate(Sender: TObject);
begin
  db:=TjanSQL.create;
end;


procedure TjanSQLDemoF.FormDestroy(Sender: TObject);
begin
  db.free;
end;
```

TjanRecordSet Overview

TjanRecordSet Overview

TjanRecord Overview

TjanRecord Overview

TjanRecordSetList Overview

TjanRecordSetList Overview

TjanRecordList Overview

TjanRecordList Overview

TjanSQLExpression2 Overview

TjanSQLExpression2 Overview
This expression evaluator semi-compiles the expression that you assign,   allowing for very fast evaluation.

This component was derived from several other evaluators I wrote in the past. It includes InFix to PostFix conversion.
It is tailered for use with TjanSQL, but can be modified for general purpose use.

The evaluator uses only a single data type: variant, and uses a stack for calculations.

When you assign an expression it is tokenized using the TjanSQLTokenizer tokenizer.

SQL introduction

SQL introduction

janSQL supports only a subset of standard SQL but the supported statements are sufficient for single-user desktop application.

table updates

janSQL loads tables automatically into memory when needed by a query. Any changes to tables are performed in memory. Tables are saved to disk when you use the COMMIT statement. The only exceptions to this are the CREATE TABLE statement, where the new table is saved to disk immediately and the DROP TABLE statement, where the table is immediately deleted from both memory and disk.

Indexes

janSQL does not use indexes. You will find that for single-user desktop applications running in memory there is no urgent need for indexes.

Case sensitivity

janSQL is case-insensitive for its keywords: you can use both SELECT and select.

Non-standard

janSQL has several non-standard SQL statements for manipulation of recordsets.

- ASSIGN TO
- SAVE TABLE
- RELEASE TABLE

Compound Queries

By using the non-standard ASSIGN TO statement you can store the result of a select query as a named variable that can be used in subsequent queries.

CONNECT TO

CONNECT TO

Connects to a database. In janSQL a database is a folder. Tables are stored in this folder as ; delimited text files with the .csv extension.

Syntax:

```
CONNECT TO 'absolute-folder-path'
```

Example:

```
connect to 'G:'
```

Notes

This is always the first statement that you use with janSQL. All other SQL statements require that the engine knows which folder to use.

COMMIT

COMMIT
Allows you to save modified in-memory tables to disk.

Syntax:

```
COMMIT
```

In janSQL all data handling is done in-memory and nothing is saved to disk until you issue the COMMIT command.

Whenever you make a change to a table with ALTER TABLE, UPDATE or DELETE, the change flag of the recordset is set. Only recordsets that have the change flag set will be saved. The change flag is reset after saving.

Only persistent recordsets are saved. A persistent recordset is a table loaded from disk.

CREATE TABLE

CREATE TABLE

Creates a new table in the current catalog.

Syntax:

```
  CREATE TABLE tablename (field1,[fieldN])
```

Example:

```
  CREATE TABLE users (userid,username,accountname, accountpassword)
```

Notes

janSQL does not use fieldtypes. Everything is stored as text. Internally janSQL treats all data as variants. This means that in your SQL queries you can use fields pritty much the way you want to.

DROP TABLE

DROP TABLE

Drops a table from the database.

Syntax:

```
  DROP TABLE tablename
```

Syntax:

```
  DROP TABLE users
```

Notes

Use with care.

INSERT INTO

INSERT INTO

Allows you to insert data in a table, either row by row or from a recordset resulting from a SELECT.

Syntax:

```
  INSERT INTO tablename [(column1[,column])] VALUES (field1[,fieldN])
  INSERT INTO tablename selectstatement
```

Example:

```
  INSERT INTO users VALUES (600,'user-600');
  INSERT INTO users (userid,username) VALUES (601,'user-601');

  INSERT INTO users SELECT * FROM users WHERE userid>400
```

Notes

When you insert records using a sub select you must make sure that the output fields of the sub select match the fieldnames of tablename. Only values of matching field will be inserted.

SELECT FROM

SELECT FROM
Allows you to select data from one or two tables.

Syntax:

```
SELECT fieldlist FROM tablename

SELECT fieldlist FROM tablename WHERE condition

SELECT fieldlist FROM tablename1 [alias1], tablenameN [aliasN]

SELECT fieldlist FROM tablename1 [alias1], tablenameN [aliasN] WHERE condition
```

fieldlist can be * for selecting all fields or field1[,fieldN]

field: fieldname [AS fieldalias]

condition see the WHERE topic.

Notes
When you join two or more tables you must use fully qualified field names: tablename.fieldname in the WHERE clause. Both tablenames and fieldnames can be aliased.

```
SELECT u.userid as mio, u.username as ma, p.productname as muu
FROM users u,products p
WHERE u.productid=p.productid
```

Using a table alias can save you typing.

```
select products.productname as product,count(users.userid) as quantity
from users,products
where users.productid=products.productid
group by product
having quantity>10
order by product desc
```

The example above shows you that in the WHERE clause you refer to source tables (e.g. products.productid) where as in the GROUP BY, HAVING and ORDER BY clause, you refer to the result table.

Always use an aliased field name when using an aggregate function:

```
count(users.userid) as quantity
```

WHERE

WHERE

The WHERE clause can be used together with the SELECT, UPDATE and DELETE clauses.

Syntax:

```
  WHERE condition
```

condition

The condition is an expression that must evaluate to a boolean true or false. The following operators are allowed:

Arithmetic

```
+ - * / ( )
```

Logic

```
and or
```

comparison

```
< <= = > >=
```

string constants

```
e.g. 'Jan Verhoeven'
```

numeric constans

```
e.g. 12.45
```

fieldnames

```
e.g. userid, users.userid
```

IN

```
e.g.
   userid IN (300,401,402)
   username IN ('Verhoeven','Smith')
```

Like

```
e.g.
  username Like '%Verhoeven'
```

You can use the % character to match any series of characters:

```
  '%Verhoeven' will match Verhoeven at the end of username
  'Verhoeven%' will match Verhoeven at the beginning of username
  '%Verhoeven%' will match Verhoeven anywhere in username
```

Sub queries

You can use a subquery after the IN clause. Only non-correlated sub queries are allowed at the moment. A sub query must select a single field from a table. A sub query is executed at parsing time and returns a comma seperated list of values that replaces the query text in the IN clause. A sub query must be enclosed by brackets.

Example:

```
select * from users where productid in (select product id from products where
productname like 'Ico%')
```

Notes

When using a SELECT with a join between 2 tables you must use fully qualified names (tablename.fieldname in every part of the query. In all other cases you must use the short form fieldname without the tablename.

UPDATE

UPDATE
Allows you to update existing data.

Syntax:

```
  UPDATE tablename SET updatelist [WHERE condition]
```

updatelist
field1=value1[,fieldN=valueN]

condition see WHERE   for the optional condition

DELETE FROM

DELETE FROM
Allows you to delete data.

Syntax:

```
  DELETE FROM tablename WHERE condition
```

condition see WHERE clause for the condition.

ALTER TABLE

ALTER TABLE

Allows you to alter the structure of a table.

Syntax:

```
  ALTER TABLE ADD COLUMN columnname
  ALTER TABLE DROP COLUMN columnname
```

You can only add or drop one column at the time.

GROUP BY

GROUP BY

Allows you to group data according grouping fields.

Syntax:

```
  group by fieldlist
```

fieldlist is a comma seperated list of one or more fields that you want to grouping to be applied.

Example:

```
  select count(userid), username, productid
  from users
  group by productid
  order by productid
```

Aggregate functions

You can apply the count, sum, avg, max, min, stddev function to an input field. When you use these functions without a GROUP BY clause, the resultset will contain only one row.

HAVING

HAVING

Allows you to filter a recordset resulting from a GROUP BY clause.

Syntax:

```
 HAVING expression
```

Example:

```
  select count(userid), username, productid
  from users
  group by productid
  having userid>10
  order by productid
```

Notes

Experienced SQL users will notice that janSQL uses a non-standard syntax in the HAVING clause. Instead of the standard having count(userid)>10, in janSQL you just use the name of the base table field, in this case userid.

You should be aware of the difference between the WHERE clause and the HAVING clause. The WHERE clause is applied to table(s) in the FROM clause. The HAVING is applied after filtering with where and grouping with group by have been applied. The same applies to the ORDER BY clause wich is also applied to the final result set.

ORDER BY

ORDER BY
Allows you to sort the resulting recordsets.

Syntax:

```
   ORDER BY orderlist
```

Example:

```
   select * from users order by #userid asc, productid desc
```

orderlist is a comma seperated list of one or more order by components

```
  component1[,componentN]
```

ordercomponent:
`[#]fieldname [ASC|DESC]`

By placing the optional # before a fieldname it will be treated as a numeric field in the sort. Remember that in janSQL all data is stored as text.

After the fieldname you can optionally put ASC for an ascending sort, or DESC for a descending sort. When you omit the sort direction the default ascending sort order is used.

ASSIGN TO

ASSIGN TO
Allows you to assign the result of a SELECT statement to a named recordset that can be referred to in subsequent statements. This is a non-standard
SQL statement. ASSIGN TO is like a variable assignment. You can create very complex compound queries with ASSIGN TO.

Syntax:

```
  ASSIGN TO tablename selectstatement
```

Example:

```
  ASSIGN TO mis SELECT userid, username FROM users
```

If tablename allready exists in the catalog then an error occurs.

When tablename does not exist in the catalog but was allready assigned to then the existing recordset is overwritten.

Notes
Make sure that you use output field alias names when you ASSIGN TO using a SELECT with joined tables.

janSQL always creates a new recordset when you execute a SELECT statement. The janSQLDemo program will release this recordset after the resultset is displayed. When you execute the ASSIGN TO the given name will be assigned to the new recordset and the recordset itself will not be released until you use RELEASE TABLE.

RELEASE TABLE

RELEASE TABLE
Allows you to release any open table from memory, including intermediate tables created with ASIGN TO. This is a non-standard SQL statement.

Syntax:

```
  RELEASE TABLE tablename
```

Example:

```
  ASSIGN TO mis SELECT * FROM users
  RELEASE TABLE mis
```

SAVE TABLE

SAVE TABLE

Allows you to save any open table, including intermediate tables, to a file. This is a non-standard SQL statement.

Syntax:

```
  SAVE TABLE tablename
```

When tablename is not open, an error occurs. When you save an intermediate table (created with ASSIGN TO), the intermediate table becomes a persistant table that is also saved with the COMMIT statement.

Example:

```
  ASSIGN TO mis SELECT * FROM users
  SAVE TABLE mis
```

Notes

Once you have saved an intermediate table with TABLE SAVE you can not ASSIGN TO anymore.

Functions

Functions

In janSQL you can use functions wherever you can use an expression to be calculated (Calculated output fields, WHERE clause, HAVING clause).

Use extra brackets around function parameters when you have a function with more than one parameter:

```
  SELECT trunc((userid/7),2) as foo FROM users
```

and **not**:

```
  SELECT trunc(userid/7,2) as foo FROM users
```

Conversion

fix(expression,precision)
Returns the string presentation of (numeric) expression with precision number of decimals.

```
  select fix((userid/7), 2) as bobo from users order by bobo
```

You can also use TRUNC i.s.o. FIX.

asnumber(expression
Returns (number or string) expression as number. If expression is not a valid floating point number then the function returns 0.

FORMAT function

FORMAT function

Formats a integer or floating point value to a string in a way specified by a format string.

Syntax:

```
  format(value,formatstring)
```

Example:

```
  update users set userid=format(userid,'%.8d')
```

Format strings have the following form:

```
[literalstring]"%" [width] ["." prec] type
```

- An optional literal string that is copied to the output
- An optional width specifier, [width]
- An optional precision specifier, ["." prec]
- The conversion type character, type

The following table summarizes the possible values for type:

d

Decimal. The argument must be an integer value. The value is converted to a string of decimal digits. If the format string contains a precision specifier, it indicates that the resulting string must contain at least the specified number of digits; if the value has less digits, the resulting string is left-padded with zeros.

u

Unsigned decimal. Similar to 'd' but no sign is output.

e

Scientific. The argument must be a floating-point value. The value is converted to a string of the form "-d.ddd...E+ddd". The resulting string starts with a minus sign if the number is negative. One digit always precedes the decimal point.The total number of digits in the resulting string (including the one before the decimal point) is given by the precision specifier in the format string—a default precision of 15 is assumed if no precision specifier is present. The "E" exponent character in the resulting string is always followed by a plus or minus sign and at least three digits.

f

Fixed. The argument must be a floating-point value. The value is converted to a string of the form "-ddd.ddd...". The resulting string starts with a minus sign if the number is negative.The number of digits after the decimal point is given by the precision specifier in the format string—a default of 2 decimal digits is assumed if no precision specifier is present.

g

General. The argument must be a floating-point value. The value is converted to the shortest possible decimal string using fixed or scientific format. The number of significant digits in the resulting string is given by the precision specifier in the format string—a default precision of 15 is assumed if no precision specifier is present.Trailing zeros are removed from the resulting string, and a decimal point appears only if necessary. The resulting string uses fixed point format if the number of digits to the left of the decimal point in the value is less than or equal to the specified precision, and if the value is greater than or equal to 0.00001. Otherwise the resulting string uses scientific format.

n

Number. The argument must be a floating-point value. The value is converted to a string of the form "-d,ddd,ddd.ddd...". The "n" format corresponds to the "f" format, except that the resulting string contains thousand separators.

m

Money. The argument must be a floating-point value. The value is converted to a string that represents a currency amount. The conversion is controlled by the CurrencyString, CurrencyFormat, NegCurrFormat, ThousandSeparator, DecimalSeparator, and CurrencyDecimals global variables, all of which are initialized from the Currency Format in the International section of the Windows Control Panel. If the format string contains a precision specifier, it overrides the value given by the CurrencyDecimals global variable.

s

String. The argument must be a string value. The string is inserted in place of the format specifier. The precision specifier, if present in the format string, specifies the maximum length of the resulting string. If the argument is a string that is longer than this maximum, the string is truncated.

x

Hexadecimal. The argument must be an integer value. The value is converted to a string of hexadecimal digits. If the format string contains a precision specifier, it indicates that the resulting string must contain at least the specified number of digits; if the value has fewer digits, the resulting string is left-padded with zeros.

Date functions

Date functions

Several date functions make working with date strings easier.

YEAR

Extracts the integer year part of a yyyy-mm-dd date string.

MONTH

Extracts the integer month part of a yyyy-mm-dd date string.

DAY

Extracts the integer day part of a yyyy-mm-dd date string.

WEEKNUMBER

Returns the integer weeknumber of a yyyy-mm-dd date string.

EASTER

Returns the easter yyyy-mm-dd date string of a given integer year.

DATEADD

Adds a given number of time intervals to a given data and returns the resulting data as a yyyy-mm-dd data string.

Syntax:

```
  DATEADD(interval,number,datestring)
```

Interval can be: 'd' (day), 'm' (month), 'y' (year), 'w' (week), 'q' (quarter).
Number must be an integer number.
datestring must be in the yyyy-mm-dd format.

String Functions

String Functions
janSQL comes with a range of functions that work on strings.

soundex(expression)
Calculates the soundex value of (string) expression. Only usefull with english terms.

lower(expression)
Converts (string) expression to lower case.

upper(expression)
Converts (string) expression to upper case.

trim(expression)
Trims (string) expression from leading and trailing spaces.

left(expression,count)
Returns the first count characters of expression

right(expression,count)
Returns the last count characters of expression

mid(expression,from,count)
Returns count characters of expression starting at from.

length(expression)
Returns the length of (string) expression. Can be used to e.g. select fields that exceed a given length.

replace(source,oldpattern,newpattern
Replaces oldpattern with new pattern in the source string. Is case-insensitive.

```
UPDATE users SET username=replace(username,'user-','foo-')
```

substr_after(source,substring)
Returns the part of source that comes after substring. If substring is not found an empty string is returned.

substr_before(source,substring)
Returns the part of source that comes before substring. If substring is not found an empty string is returned.

Numeric Functions

Numeric Functions
Numeric functions work on strings as if they were numbers. Although janSQL is based on strings you can still enter values like 1234, which can be treated like numbers.

Numeric

sqr(expression
Calculates the square of (numeric) expression.

sqrt(expression
Calculates the square root of (numeric) expression.

sin(expression
Calculates the sin of (numeric) expression.

cos(expression
Calculates the cos of (numeric) expression.

ceil(expression
Returns the lowest integer greater than or equal to (numeric) expression.

floor(expression
Returns the the highest integer less than or equal to (numeric) expression.