(Un)Perplexed Spready - SQL User Manual

Ver 1.0

2025-03-02

---

1. Introduction

(Un)Perplexed Spready SQL is a powerful single-user SQL database management system integrated directly into your spreadsheet environment. Unlike traditional database systems, it requires **no server installation** and **no external libraries**. It is a self-contained, simple, and effective solution that can process and query data from your spreadsheet sheets using standard SQL syntax.

(Un)Perplexed Spready SQL is based on the proven **janSQL engine** (by Jan Verhoeven, http://jansfreeware.com), the same engine used in ZMSQL package and MightyQuery software.

If you need to:

- Process heterogeneous business data from various ERP, MRP, Data Warehouse, or Accounting systems
- Perform complex data analysis directly within your spreadsheet
- Join multiple sheets, filter data, calculate aggregates
- Transform and export data between different formats

(Un)Perplexed Spready SQL is your solution.

Key Features

- **Workbook-as-Database**: Each spreadsheet sheet becomes a SQL table
- **No Registration Required**: Sheets are automatically available as SQL tables
- **In-Memory Processing**: Complete in-memory handling of tables and recordsets for high performance
- **Semi-Compiled Expressions**: Optimized expression evaluation
- **Visual Query Builder**: Build queries without writing SQL
- **Syntax Highlighting**: Full SQL syntax highlighting in the editor
- **Auto-Complete**: Intelligent SQL keyword completion
- **Export Capabilities**: Export results to new sheets

---

2. General Concepts

2.1 Workbook-as-Database Architecture

(Un)Perplexed Spready treats your entire workbook as a **database catalog**, where:

- **Each sheet** = One SQL table
- **Sheet name** = Table name
- **First row** = Column headers (field names)
- **Subsequent rows** = Data records

Unlike MightyQuery where you must register database folders and textual files, in (Un)Perplexed Spready **all sheets in the current workbook are immediately available as SQL tables**. Simply open a workbook, open the SQL Query window, and start querying.

2.2 Data Storage

Data is stored in the spreadsheet itself (.xlsx, .ods, .csv):

- Sheets serve as tables
- No external file registration is needed
- Changes can be saved via COMMIT or by exporting results to new sheets

2.3 Data Types

Unlike most database engines that have strong data typing, **janSQL does not use data types**. 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 strings in the **ISO 8601 format**:

- `YYYY-MM-DD` (e.g., 2025-03-02)
- `YYYY-MM-DDThh:mm:ss` (e.g., 2025-03-02T14:30:00)

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:

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

Note: 1953-11-16 will be less than 2002-03-26 when compared as strings.

It is advised to store dates and times in separate fields:

- **Dates**: `YYYY-MM-DD` format
- **Times**: `hh:mm:ss` format (00:00:00 to 23:59:59)

2.4 Organization and Terminology

(Un)Perplexed Spready SQL is organized as a single integrated **SQL Query window** that contains all functionalities:

- **SQL Editor** - Where you enter and edit SQL statements
- **Results Grid** - Where query results are displayed
- **Toolbar** - Quick access to execute, export, save/load queries

- **Query Builder** - Visual query construction tool

Table Naming:

- Tables are named exactly as the corresponding sheet name
- If a sheet is renamed, the table name changes accordingly
- All sheets are automatically available - no registration needed

---

3. Accessing SQL Functionality

3.1 Opening the SQL Query Window

1. Open (Un)Perplexed Spready
2. Load or create a workbook with data
3. Click **Tools** → **SQL Query** from the menu bar
4. The SQL Query window will open with:
    - **SQL Editor** (top) - Where you write queries
    - **Results Grid** (bottom) - Where results are displayed
    - **Toolbar** - Quick access buttons

3.2 SQL Window Components

Toolbar Buttons:

- **Execute (F5)** - Run the SQL query
- **Clear** - Clear the SQL editor
- **Export to Sheet** - Export results to a new spreadsheet sheet
- **Save Query** - Save SQL to a file (.sql)
- **Load Query** - Load SQL from a file
- **Query Builder** - Launch visual query builder

SQL Editor Features:

- Syntax highlighting for SQL keywords
- Auto-complete (Ctrl+Space)
- Line numbers
- F5 or Ctrl+Enter to execute

---

4. Writing Changes to Sheets

All data processing is done in memory, rather than directly on the spreadsheet. When you execute SQL commands that modify data (INSERT, UPDATE, DELETE), changes are made to an in-memory copy of the table.

To persist changes:

1. Use the **COMMIT** statement to save all modified tables back to the workbook
2. Use **SAVE TABLE** to save a specific result set as a new sheet
3. Use **Export to Sheet** button to save query results to a new sheet

Important: Unless you issue COMMIT or use SAVE TABLE/Export, changes remain in memory only and are lost when you close the SQL window or the application.

---

5. SQL Editor

In the SQL editor you can enter or view SQL expressions for the current query.

5.1 Entering SQL

You can enter or modify SQL text manually. The SQL editor has:

- **Code coloring** (syntax highlighting)
- **Autocompletion** feature - Press [CTRL]+[SPACE] to open a drop-down list for autocompletion

5.2 Helper Lists

The SQL window provides helper lists of:

- Available tables (sheets)
- Fields (columns) from selected table
- SQL keywords and expressions
- Operators

By selecting items from these lists and clicking "Apply", you can copy the selected item to the current cursor position in the SQL editor.

5.3 Saving and Loading Queries

Save Query to File:

1. Click **Save Query** button
2. Provide file name (with .sql extension recommended)
3. Query is saved as a text file

Load Query from File:

1. Click **Load Query** button
2. Select the .sql file
3. Query is loaded into the editor

---

6. Executing Queries

6.1 Running Queries

Click the **Execute** button (or press F5) to run the SQL statement.

Results are displayed in the Results Grid below the editor. You can:

- Navigate through records
- View all returned columns
- See the number of rows and execution time in the status bar

6.2 Exporting Results

To save query results to a new sheet:

1. Click **Export to Sheet** button
2. Enter a name for the new sheet
3. Results are saved as a new sheet in the current workbook

The exported sheet can then be used as a regular table in subsequent queries.

---

7. SQL Introduction

The janSQL database engine supports a subset of standard SQL with some powerful extensions. All SQL keywords are **case-insensitive** - you can use SELECT or select.

7.1 Table Updates

janSQL loads tables (sheets) automatically into memory when needed by a query. Any changes to tables (INSERT, UPDATE, DELETE) are performed in memory. Tables are saved to the workbook when you use the **COMMIT** statement.

Exceptions:

- **CREATE TABLE**: The new sheet is saved immediately
- **DROP TABLE**: The sheet is immediately deleted from both memory and the workbook

7.2 Indexes

janSQL does not use indexes. For single-user desktop applications running in memory, there is no urgent need for indexes.

7.3 Case Sensitivity

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

7.4 Non-Standard Extensions

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

- **ASSIGN TO** - Create named temporary tables
- **SAVE TABLE** - Persist a table to a sheet
- **RELEASE TABLE** - Remove a table from memory

7.5 Compound Queries

You can execute multiple queries in a batch. Query expressions must be separated by semicolon (;).

Example:

```
ASSIGN TO temp SELECT * FROM Orders WHERE Amount > 1000;
SELECT * FROM temp ORDER BY Date;
RELEASE TABLE temp
```

---

8. SQL Syntax Reference

A) Data Definition Language

CONNECT TO

Connects to a database. In janSQL, a database is a folder containing CSV/text files. This is primarily for importing external data.

Syntax:

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

Example:

```
CONNECT TO '/home/user/data'
```

Note: When working with the current workbook, you typically don't need CONNECT TO - all sheets are automatically available as tables.

CREATE TABLE

Creates a new sheet in the current workbook.

Syntax:

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

Example:

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

Note: janSQL does not use field types. Everything is stored as text. Internally, janSQL treats all data as variants, meaning you can use fields pretty much the way you want to.

DROP TABLE

Drops a sheet from the workbook.

Syntax:

```
DROP TABLE tablename
```

Example:

```
DROP TABLE users
```

Warning: Use with care, because it permanently deletes the sheet and all its data!

ALTER TABLE

Allows you to alter the structure of a sheet.

Syntax:

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

Note: You can only add or drop one column at a time.

B) Data Manipulation Language

SELECT FROM

Allows you to select data from one or more sheets.

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:

- * for selecting all fields
- field1[,fieldN] for specific fields
- fieldname [AS fieldalias] for aliasing

Examples:

```
-- Select all columns
SELECT * FROM Customers

-- Select specific columns with aliases
SELECT u.userid as mio, u.username as ma, p.productname as muu
FROM users u, products p
WHERE u.productid = p.productid

-- Complex join example with multiple fields
SELECT p.PRDCT as Product, p.PRDCT_DSCR as ProductDescription,
      p.PRDCT_BASE_QTY as ProductBaseQuantity,
      p.PRDCT_BASE_QTY_UNT as ProductUOM,
      b.CMPNT as Component, b.CMPNT_DSCR as ComponentDescription,
      b.CMPNT_BRTT_QTY as ComponentBruttoQuantity,
      b.CMPNT_BRTT_QTY_UNT as ComponentUOM
FROM products p, boms b
WHERE p.PRDCT = b.PRDCT
AND p.PRDCT = '895104'
```

Notes:

- When joining two or more tables, you must use fully qualified field names: tablename.fieldname in the WHERE clause
- Both tablenames and fieldnames can be aliased
- Using table aliases can save you typing

```
-- Example with joins, aggregates, and ordering
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
```

Important: In the WHERE clause you refer to source tables (e.g., products.productid), whereas in the GROUP BY, HAVING and ORDER BY clauses, you refer to the result table (e.g., product).

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

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

WHERE

The WHERE clause can be used with SELECT, UPDATE and DELETE statements.

Syntax:

```
WHERE condition
```

Condition is an expression that must evaluate to boolean true or false.

Allowed Operators:

Type	Operators
Arithmetic	+-*/()
Logic	and, or
Comparison	<<==>>=<>

String constants: Use single quotes: 'Jan Verhoeven'

Numeric constants: Direct numbers: 12.45

Fieldnames: Direct names or qualified: userid, users.userid

IN Operator:

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

LIKE Operator:

```
username LIKE '%Verhoeven'
username LIKE 'Verhoeven%'
username LIKE '%Verhoeven%'
```

Use `%` as a placeholder to match any series of characters:

- `'%Verhoeven'` matches Verhoeven at the end
- `'Verhoeven%'` matches Verhoeven at the beginning
- `'%Verhoeven%'` matches Verhoeven anywhere

IMPORTANT NOTE: Sometimes the janSQL engine cannot process WHERE clauses with several conditions and joins. The application may freeze in such cases. Therefore, try to simplify the WHERE clause by filtering one of the joined tables prior to the main expression. Split the main query into multiple steps using **ASSIGN TO**:

```
-- Instead of complex single query, use intermediate steps
ASSIGN TO filtered_users SELECT * FROM users WHERE status = 'Active';
SELECT * FROM filtered_users u, orders o WHERE u.id = o.userid;
RELEASE TABLE filtered_users
```

Subqueries

You can use a subquery after the IN clause. Only **non-correlated subqueries** are supported. A subquery must select a single field from a table. It is executed at parsing time and returns a comma-separated list of values.

Example:

```
SELECT * FROM users
WHERE productid IN (SELECT productid FROM products WHERE productname LIKE 'Ico%')
```

Notes:

- A subquery must be enclosed by brackets
- When using SELECT with a join between 2 tables, use fully qualified names (`tablename.fieldname`)
- In all other cases (UPDATE, INSERT), use the short form `fieldname` without the tablename

---

GROUP BY

Allows you to group data according to grouping fields.

Syntax:

```
GROUP BY fieldlist
```

Example:

```
SELECT count(userid), username, productid
FROM users
GROUP BY productid
ORDER BY productid
```

Aggregate Functions:

You can apply these functions to an input field:

- COUNT(field) - Count records
- SUM(field) - Sum of values
- AVG(field) - Average value
- MAX(field) - Maximum value
- MIN(field) - Minimum value
- STDDEV(field) - Standard deviation

When you use these functions without a GROUP BY clause, the resultset will contain only one row.

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
```

Important: 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: `having userid>10`.

Difference between WHERE and HAVING:

- **WHERE** is applied to table(s) in the FROM clause
- **HAVING** is applied after filtering with WHERE and grouping with GROUP BY
- **ORDER BY** is also applied to the final result set

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-separated list of one or more order components:

```
[#]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, so use `#` for numeric sorting

- `ASC` = ascending (default), `DESC` = descending

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.

Syntax:

```
ASSIGN TO tablename SELECT statement
```

Example:

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

If tablename already exists in the catalog, an error occurs. When tablename does not exist in the catalog but was already assigned to, the existing recordset is overwritten.

Notes:

- Make sure that you use output field alias names when using ASSIGN TO with SELECT with joined tables
- The recordset is not released until you use RELEASE TABLE

RELEASE TABLE

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

Syntax:

```
RELEASE TABLE tablename
```

Example:

```
ASSIGN TO mis SELECT * FROM users;
-- use mis in other queries
RELEASE TABLE mis
```

SAVE TABLE

Allows you to save any open table, including intermediate tables, to a sheet. 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), it becomes a persistent sheet.

Example:

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

Note: Once you have saved an intermediate table with SAVE TABLE, you cannot ASSIGN TO it anymore.

INSERT INTO

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

Syntax:

```
INSERT INTO tablename [(column1[,columnN])] VALUES (field1[,fieldN])
INSERT INTO tablename SELECT statement
```

Examples:

```
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 inserting records using a sub-select, ensure the output fields of the sub-select match the fieldnames of tablename
- Only values of matching fields will be inserted

UPDATE

Allows you to update existing data.

Syntax:

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

Updatelist:

```
field1=value1[,fieldN=valueN]
```

Example:

```
UPDATE users SET username = 'newname', status = 'Active' WHERE userid = 100
```

DELETE FROM

Allows you to delete data.

Syntax:

```
DELETE FROM tablename [WHERE condition]
```

Example:

```
DELETE FROM users WHERE status = 'Inactive'
```

Note: Without a WHERE clause, all records are deleted (use with extreme caution).

COMMIT

Saves all modified tables to the workbook.

Syntax:

```
COMMIT
```

Execute this command after INSERT, UPDATE, or DELETE operations to persist changes to the spreadsheet.

C) Functions

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

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

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

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

Conversion Functions

FIX(expression, precision) or **TRUNC(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
```

ASNUMBER(expression)

Returns (number or string) expression as a number. If expression is not a valid floating point number, returns 0.

FORMAT(value, formatstring)

Formats an integer or floating point value to a string according to the format string.

```
UPDATE users SET userid = format(userid,'%.8d')
```

Format String Syntax:

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

- Optional literal string copied to output
- Optional width specifier
- Optional precision specifier
- Conversion type character

Type Characters:

Type	Description
d	Decimal integer
u	Unsigned decimal
e	Scientific notation
f	Fixed decimal
g	General (shortest format)
n	Number with thousand separators
m	Currency/money format
s	String (with max length)
x	Hexadecimal

Date Functions

YEAR(datestring)

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

MONTH(datestring)

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

DAY(datestring)

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

WEEKNUMBER(datestring)

Returns the integer week number of a `yyyy-mm-dd` date string.

EASTER(year)

Returns the Easter `yyyy-mm-dd` date string for a given integer year.

DATEADD(interval, number, datestring)

Adds a given number of time intervals to a date.

Interval values:

- `'d'` - day
- `'m'` - month
- `'y'` - year
- `'w'` - week
- `'q'` - quarter

```
SELECT DATEADD('d', 7, '2025-03-02')  -- Adds 7 days
```

String Functions

SOUNDEX(expression)

Calculates the soundex value of a string (useful for English terms).

LOWER(expression)

Converts string to lowercase.

UPPER(expression)

Converts string to uppercase.

TRIM(expression)

Removes leading and trailing spaces.

LEFT(expression, count)

Returns the first count characters.

RIGHT(expression, count)

Returns the last count characters.

MID(expression, from, count) or **SUBSTRING(expression, from, count)**

Returns count characters starting at position from.

LENGTH(expression) or **LEN(expression)**

Returns the length of the string.

REPLACE(source, oldpattern, newpattern)

Replaces old pattern with new pattern (case-insensitive).

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

SUBSTR_AFTER(source, substring)

Returns the part of source that comes after substring. Returns empty string if not found.

SUBSTR_BEFORE(source, substring)

Returns the part of source that comes before substring. Returns empty string if not found.

Numeric Functions

SQR(expression)

Calculates the square.

SQRT(expression)

Calculates the square root.

SIN(expression)

Calculates the sine.

COS(expression)

Calculates the cosine.

CEIL(expression)

Returns the lowest integer greater than or equal to the expression.

FLOOR(expression)

Returns the highest integer less than or equal to the expression.

---

9. Tips and Best Practices

Performance Tips

1. **Simplify complex WHERE clauses** with multiple joins by using ASSIGN TO
2. **Use # prefix** for numeric fields in ORDER BY: `ORDER BY #userid`
3. **Limit results** with WHERE clauses before complex operations
4. **Release temporary tables** with RELEASE TABLE when done

Data Quality

1. Store dates in **ISO 8601 format** (`YYYY-MM-DD`) for proper sorting
2. Be aware that **everything is stored as text** - use conversion functions when needed
3. Use **TRIM()** when comparing text fields that may have whitespace
4. Use **ISNUMERIC()** or **ASNUMBER()** before numeric calculations

Common Patterns

Find duplicates:

```
SELECT Email, COUNT(*) as Count
FROM Customers
GROUP BY Email
HAVING Count > 1
```

Top N records:

```
SELECT * FROM Sales ORDER BY #Amount DESC LIMIT 10
```

Date range filter:

```
SELECT * FROM Orders
WHERE Date >= '2025-01-01' AND Date <= '2025-02-28'
```

Clean up temporary tables:

```
ASSIGN TO temp SELECT * FROM LargeTable WHERE Condition;
-- work with temp
RELEASE TABLE temp
```

---

10. Technical Information

10.1 Based On

- **janSQL Engine** - Original code by Jan Verhoeven (jansfreeware.com)
- **ZMSQL Plus Package** - Enhanced version for CodeTyphon/Lazarus
- **fpSpreadsheet** - Spreadsheet file format support
- **SynEdit** - Syntax highlighting and auto-completion

10.2 License

The janSQL engine is released under the Mozilla Public License 1.1.

Modified source code is included in the installation folder.

Happy Querying with (Un)Perplexed Spready SQL!