DUCK VBA DLL

Etienne Lenoir

February 5, 2026

Excel/VBA & Access: Upgraded with DuckDB

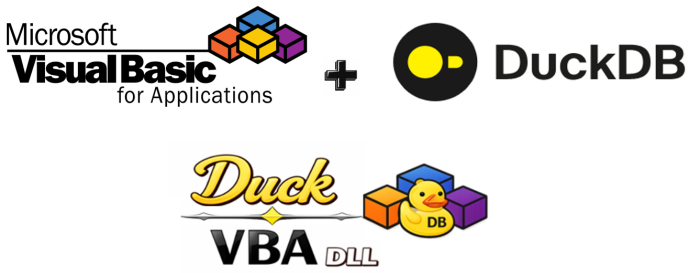
Duck VBA DLL — C/C++ DLL Kit & VBA Toolkit

Contents

1. Introduction

1.1 Duck VBA DLL overview

Duck-VBA-DLL integrates DuckDB into Excel/VBA: a highly performant local engine, with no installation and no external dependencies, and a modern OLAP database for analytical processing—an excellent alternative to Microsoft Access.



Duck-VBA is a native bridge (**Windows DLL in C/C++**) between **VBA** (Excel/Office) and **DuckDB**, an embedded analytical SQL engine. The goal is simple: give VBA an **ultra-fast compute engine** that is easy to deploy (a single `.dll` next to the `.xlsm`).

With Duck-VBA, Excel remains the front-end (data entry, controls, reporting), while DuckDB becomes the local back-end for everything that is expensive in time and complexity: JOIN, GROUP BY, window functions (WINDOW), CTEs, upserts, staging, transactions, imports/exports (CSV/Parquet/JSON), and large-scale processing.

Concretely, it:

- Loads data into an embedded database (from Excel, CSV, Parquet, JSON…)
- Processes and transforms with SQL inside DuckDB (vectorized, optimized, multi-threaded engine)
- Handles millions of rows

Everything works either in persistent mode (a local, portable `.duckdb` file), or in ephemeral `:memory:` mode (in **RAM**) for ultra-fast pipelines with no disk I/O. The result: VBA finally "digests" data like a modern data tool, with a **dataframe-like** approach (pandas-style) driven by SQL.

The DLL upgrades VBA by providing a faster, more robust, more modern path—by offloading data operations to DuckDB's optimized SQL engine (vectorized, multi-threaded).

1.2 Context

VBA is still widely used in companies because it is native to Office, already deployed everywhere, easy to share via a simple `.xlsm`, and perfect for the "last mile" (reporting, data entry, automation, legacy macros). Even with Python/BI, Excel remains the universal interface.

As soon as you move to "serious" data (volume, ETL, advanced SQL, transactions), traditional Excel/VBA approaches (loops, cells, ADO/ODBC) quickly become slow, fragile, and hard to industrialize. Beyond a certain threshold, limitations show up fast:

- **VBA loops**: insufficient performance once you manipulate thousands/millions of rows.
- **Power Query**: excellent for data prep, but less suited to macro-driven workflows, sometimes less transparent, and limited for transactional scenarios (staging, upsert, synchronization).
- **MS Access**: very convenient as a "local database", but it quickly caps out for modern data workloads (file size limits, less suitable for heavy transformations).

Duck-VBA addresses precisely that breaking point: it extends the "local database like Access" idea by grafting an embedded analytical SQL engine (DuckDB) into VBA—capable of absorbing heavy workloads (fast imports, temp tables, complex queries, upserts) while keeping Excel as the interface.

This is one of the first integrations enabling VBA to read and write **Parquet** directly (via DuckDB) without additional dependencies.

1.3 DuckDB overview

Website: https://duckdb.org/ • GitHub: https://github.com/duckdb/duckdb

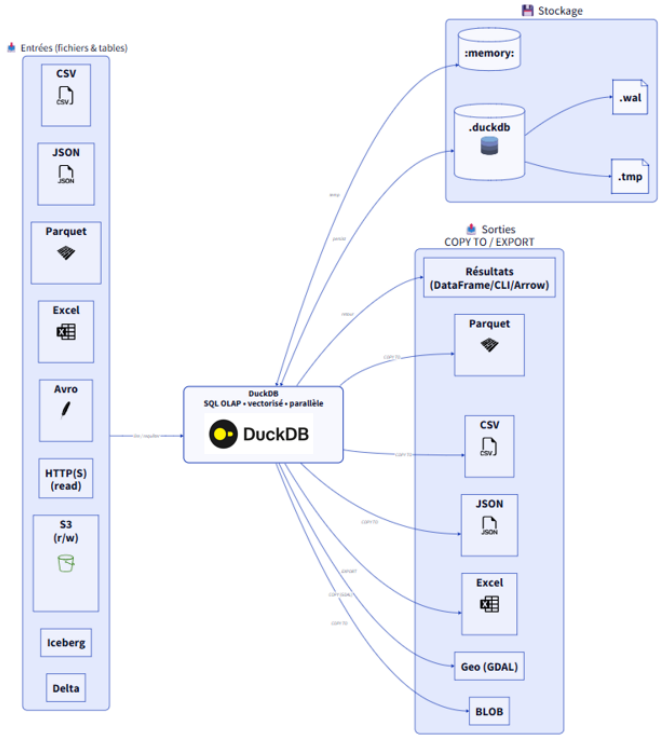
DuckDB was born at CWI[1] (Netherlands), initially developed by **Mark Raasveldt** and **Hannes Mühleisen**. The project started around **2018**, with a first public version in 2019. DuckDB is released under the MIT license.

DuckDB is an analytical SQL engine (**OLAP**) designed for **columnar processing**, built to maximize throughput on large volumes (scans, filters, aggregations, joins). Its execution is vectorized (batch processing) and parallelized (multi-threaded), which allows it to use CPU and memory efficiently for analytical queries—especially when only a subset of columns is read. It is also very well suited for columnar formats such as **Parquet** (column projection, read optimizations, pushdown).

DuckDB is **serverless**: no instance to deploy, no service to maintain, no network port to expose. It can be used either as an **in-memory** database (`:memory:`) for temporary low-latency processing, or as a persistent database through a `.duckdb` file (portable, local) to store data, metadata, and potential indexes.

It resembles **SQLite** in its embedded nature (zero server, easy to integrate, often a single file), but differs by purpose: DuckDB is optimized for analytics (OLAP, large scans and aggregations), while SQLite is mostly optimized for transactional workloads (OLTP, small CRUD operations and frequent transactions).

It reads and processes modern data formats very well (CSV, Parquet, JSON), enables advanced SQL transformations, and integrates easily into a local-first workflow. It is designed to deliver high performance on complex queries over large datasets in embedded setups, including workloads combining tables with hundreds of columns and billions of rows.



[1] Centrum voor Wiskunde en Informatica: The Dutch national research institute for mathematics and computer science in Amsterdam.

1.4 Benefits of Duck VBA DLL

The goal is not to turn VBA into a “data language”, but to give it a **modern SQL engine** for everything that is costly with Excel loops:

- joins, group by, window functions, CTEs, staging, transactions
- fast ingestion (Appender) from arrays/recordsets
- modern formats (Parquet/JSON) via DuckDB extensions
- “Excel-ready” outputs: `Variant(2D)` (direct paste) or `Dictionary` (ultra-fast lookups)

Benefits:

- ☒ A true modern database engine for VBA: embedded DuckDB in-process, serverless → fast analytical SQL (JOIN, GROUP BY, WINDOW, CTE, transactions).
- ☒ Ultra-simple deployment: one DLL + one `.xlsm`, no driver (no ODBC/DSN), no per-machine installation.
- ☒ A “local database” alternative to MS Access: portable `.duckdb` file (persistent) or `:memory:` (100% RAM) for temporary pipelines with no disk I/O.
- ☒ High-performance ingestion and modern formats: CSV / Parquet / JSON (including large volumes) with scans and imports far more efficient than classic VBA/ADO approaches.
- ☒ Optimized Excel □ SQL bridge:
  - Array / Range → DuckDB via appender (instant staging, no intermediate files)
  - SELECT → `Variant(2D)` (direct paste to sheet) or → `Dictionary` (near-instant lookups in VBA)
- ☒ “Dataframe-like” workflow (pandas-inspired): start from an in-memory Excel array, materialize it as a DuckDB frame/table (temp or persistent), apply transformations in SQL (join/group/window/CTE/casts…), then return results as `Variant(2D)` (for Excel) or as `Dictionary` (for ultra-fast lookups).
- ☒ `:memory:` mode (100% RAM): very fast pipelines without disk I/O, ideal for throwaway ETL, intermediate computations, testing, iteration—powered by a vectorized SQL engine behind your arrays.
- ☒ Ready-to-use “VBA-friendly” functions (bridge-specific):
  - Upsert from Range/Array (sync Excel → DB: update + insert)
  - Dictionary exports (flat / row1D / row2D) for mapping and fast in-memory access
  - Temp lists to replace large `WHERE IN (...)` (fast filtering on thousands of keys)
  - Scalar helpers (COUNT/MAX/…) without transferring a full table
  - Access → DuckDB bridges: import tables/recordsets, and export Access → Parquet / DuckDB depending on options (ODBC if available, otherwise ADO fallback)

1.5 What is a DLL?

A DLL (Dynamic Link Library) is a native Windows **binary** module (**PE – Portable Executable** format), similar to an `.exe`, but designed to be **loaded by another program** and provide reusable code. It contains a PE binary with machine code (opcodes) and tables (imports/exports) describing how to load and link it.

- **Link** = connect a program to a library
- **Dynamic** = the connection happens at runtime (not at compile time)

A DLL is loaded as soon as an application needs it (at startup through imports, or later on demand). It remains in memory as long as at least one program uses it: Windows maintains a reference count, and once all applications have released it, the DLL can be **unloaded**.

The main purpose of DLLs is **code and data sharing**:

- **On disk**: a single copy of the library is stored.
- **In memory**: when multiple processes use the same DLL, some pages can be shared, reducing overall RAM usage (especially for code and read-only data).

The file format of a DLL is identical to that of an executable (EXE). The main difference is that a DLL cannot be executed directly, because the OS requires an entry point to start execution. Windows provides utilities (RUNDLL.EXE/RUNDLL32.EXE) to execute a function exported by a DLL.

From C code to a DLL (what really happens)

1) Compilation (`cl.exe`)

The MSVC compiler takes the `.c` source file and produces one or more `.obj` object files.

The `.obj` already contains **native code** (x64 instructions encoded as bytes) and the information required for linking:

- symbols (functions/variables)
- sections (code, data)
- relocation information
- external references (imports to resolve)

In practice, the pipeline is: **preprocessor → parsing/type-checking → optimizations (/O2) → code generation**. The output is no longer C: it is a binary artifact ready to be assembled/linked.

> Note: even if you do not explicitly generate a `.asm` file, there is always a logical lowering step toward an assembly-like representation, then encoding into **opcodes** (machine code).

2) Linking (`link.exe`)

The linker builds `duckdb_vba_bridge.dll` from the `.obj` files and import libraries:

- `duckdb.lib` (import library for `duckdb.dll`)
- `oleaut32.lib` (COM/OLE Automation API: BSTR/SAFEARRAY/VARIANT)

It produces a **PE (Portable Executable)** DLL and builds, among other things:

- `.text`: executable code (x64 opcodes)
- `.rdata`: constants, read-only tables
- `.data`: writable data
- `.idata`: *Import Directory* + **IAT (Import Address Table)** for dependencies
- `.edata`: *Export Directory* (functions exposed to VBA/Windows)
- `.reloc`: relocations (address fixups if the image is loaded at a different base)

3) Result

The final DLL is a **native binary** containing exclusively:

- **machine code**
- data
- PE tables (imports/exports/relocations…)

There is no notion of “C” at this stage: only bytes consumable by the Windows loader and the CPU.

How Windows loads and executes a DLL

When Excel/VBA loads the DLL (directly or indirectly via `LoadLibrary`), the Windows loader performs:

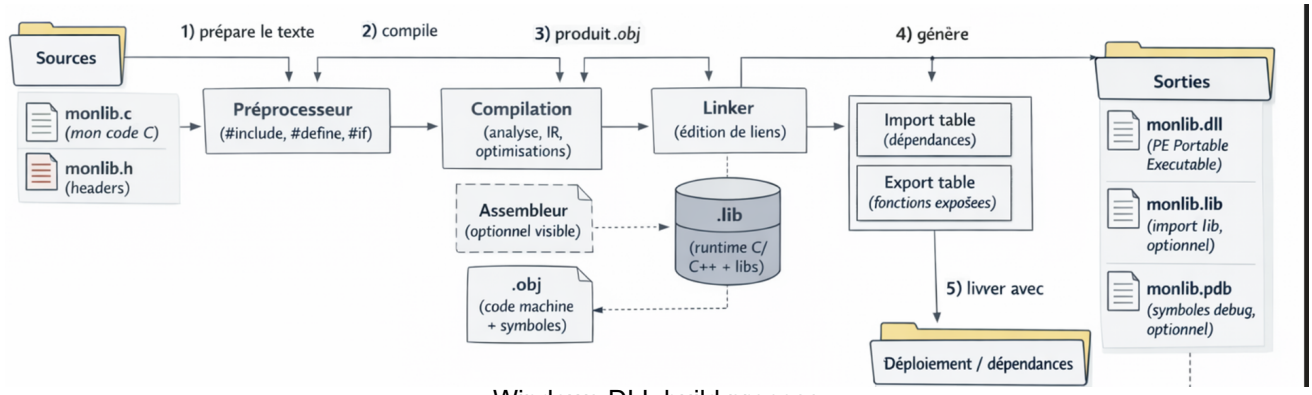
1) **Memory mapping** of the PE sections

2) Applying **memory protections**:
   - code as *Read + Execute*
   - data as *Read + Write*
3) Resolving **imports**:
   - loading dependencies (`duckdb.dll`, `oleaut32.dll`, etc.)
   - filling the **IAT** with actual function addresses
4) Applying **relocations** if the DLL is not loaded at its preferred image base
5) Optional call to `DllMain` (module initialization)

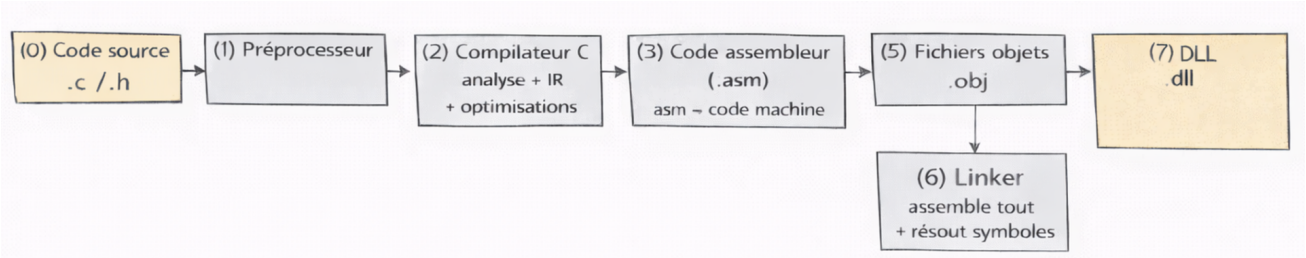
Then, when an exported function is called from VBA:

- Excel retrieves the function address (Export Table / `GetProcAddress`)
- the call becomes a simple jump to a memory address inside `.text`
- the **CPU executes** the x64 opcodes directly (*fetch* → *decode* → *execute*)

In other words: Windows “understands” a DLL through its PE format and tables; the CPU executes the code because it is already machine instructions—not interpreted code.



Windows DLL build process



Simplified pipeline: DLL compilation

2. Installation

The `duckdb_vba_bridge.dll` DLL is compiled and tested with **DuckDB v1.4.3** (December 2025).

Prerequisites:

- Windows
- Microsoft Visual C++ Redistributable (often already installed on corporate/Office machines)
  - If required, you may need the following DLLs in the same folder: `vcruntime140.dll`, `msvcp140.dll`, `vcruntime140_1.dll`.
- Excel 64-bit (VBA7)
- DuckDB runtime: `duckdb.dll` (official DuckDB binary version 1.4.3)
- DUCK VBA DLL bridge: `duckdb_vba_bridge.dll` (this project's bridge DLL)

Deployment (zero installation):

- No system-wide install required: no server, no ODBC/DSN, no driver to install.
- Portable: by default, the DLLs can be placed in the same folder as the `.xlsm` (or a subfolder in your project).
- Configurable path: using the `cDuck` class (e.g., `cDuck.Init <path>`), you can point to another binaries folder if needed (locked-down machine, shared folder, versioning, etc.).

To integrate Duck-VBA into another workbook/VBA project, the minimum requirement is to import module `mDuckNative` and class `cDuck` (they encapsulate the native API and expose the main interface).

Windows security: after copying/downloading, check that `duckdb.dll` and `duckdb_vba_bridge.dll` are not blocked by Windows (right-click → Properties → check **Unblock** → Apply). Otherwise Excel/VBA may refuse to load the DLL.

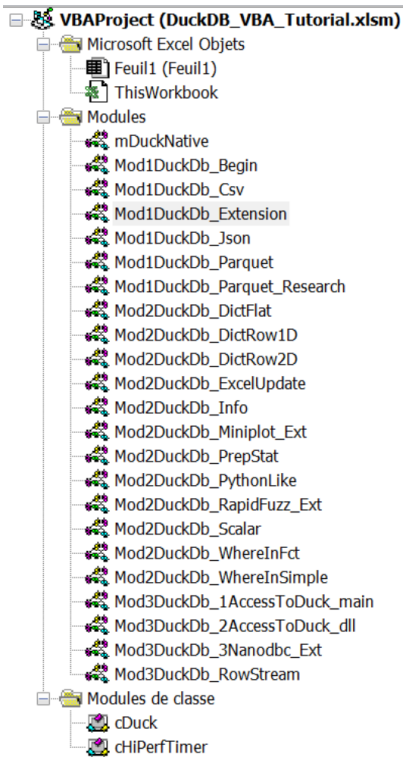
`Some features (e.g., parquet, json, odbc/rapidfuzz) may depend on DuckDB extensions.`

```
Dim db As New cDuck
'1) Init: tells DuckVba where to find the DLLs (DuckVba.dll, duckdb.dll, etc.)
'   Here: workbook folder (.xlsm). Standard if you place the DLLs next to the file.
db.Init ThisWorkbook.Path
'db.Init "C:\Tools\DuckVba"

'2) Open DuckDB:
'":memory:" = ephemeral in-RAM database (gone on close, ultra fast, zero disk I/O)
db.OpenDuckDb ":memory:"

'Variant: persistent on disk (single portable .duckdb file)
db.OpenDuckDb ThisWorkbook.Path & "\cache.duckdb"
```

3. Full tutorial (demo workbook)



A step-by-step tutorial is provided in the workbook `DuckDB_VBA_Tutorial.xlsm` It contains ready-to-run examples (imports, queries, upserts, dictionaries, parquet/json/csv, benchmarks…) and the documentation is directly inside the VBA modules (open the VBE editor: ALT + F11).

Recommendation: start with the `Begin / Info` modules and then follow the demos (procedures named `Demo_*`).

You can also consult DuckDB's official documentation, which is very complete:

https://duckdb.org/docs/stable/guides/overview

Need	`cDuck` method(s)	Demo (module)
Open/close DB	`Init`, `OpenDuckDb`, `OpenReadOnly`, `CloseDuckDb`	`Mod1DuckDb_Begin.bas`
Execute SQL	`Exec`, `ExecOK`	`Mod1DuckDb_Begin.bas`
SELECT → Excel	`QueryFast`, `SelectToSheet`	`Begin`, `Parquet`
Scalar (COUNT/…)	`Scalar*`	`Mod2DuckDb_Scalar.bas`
Load Excel array	`FrameFromValue`, `AppendArray`	`Mod2DuckDb_PythonLike.bas`
Upsert Excel → DB	`UpsertFromArray`	`Mod2DuckDb_ExcelUpdate.bas`
Large `WHERE IN`	`CreateTempList`, `SelectWithTempList`	`Mod2DuckDb_WhereIn*`
In-memory lookup	`SelectToDictFlat/Row1D/Row2D`	`Mod2DuckDb_Dict*`
Prepared statements	`Prepare`, `PS_Bind*`, `PS_Exec`	`Mod2DuckDb_PrepStat.bas`
CSV/JSON/Parquet	`ReadToArray`, `CopyToJson`, `CopyToParquet`, `ImportCsvReplace`	`Mod1DuckDb_*`
Extensions	`LoadExt`, `TryLoadExt`, `EnsureOdbcLoaded`	`Extension`, `Nanodbc`, `RapidFuzz`, `Miniplot`
Access → DuckDB	`AppendAdoRecordset*`, ODBC helpers	`Mod3DuckDb_*`
Streaming ingestion	Low-level appender	`Mod3DuckDb_RowStream.bas`

4. Getting started with Duck-VBA

This section documents **Excel/VBA usage**: how to open a DuckDB connection, execute SQL, load/extract data, and choose the right method depending on context (volume, performance, robustness, locked-down machines…).

- `mDuckNative.bas`: *foundation* (`Declare` statements, DLL loading, practical helpers).
- `cDuck.cls`: *usage-oriented façade* (error handling, high-level methods, ergonomics). One instance of `cDuck` = one DuckDB **connection** (handle).
- `db.Init(dllFolder)` calls `EnsureDuckDll` (in `mDuckNative.bas`) which:
  - sets the DLL search directory (`SetDllDirectoryW`)
  - loads `duckdb.dll` + `duckdb_vba_bridge.dll` via `LoadLibraryW`

Place `duckdb.dll` and `duckdb_vba_bridge.dll` **in the same folder** as the `.xlsm`, or in any folder you choose.

```
db.Init ThisWorkbook.Path
```

Core functions

- `OpenDuckDb(pathOrMemory)`: read/write
- `OpenReadOnly(path)`: read-only
- `CloseDuckDb`
- `Exec(sql)`: DDL/DML/COPY/PRAGMA…

Errors, logging & diagnostics

The DLL is “C-style”: most functions return a status (**1 = OK, 0 = FAIL**) and write details into an internal error buffer. The VBA wrapper (`cDuck`) standardizes this behavior into a clear, configurable error strategy.

```
Dim db As New cDuck
db.Init ThisWorkbook.Path

db.ErrorMode = 0  'DEV: Raise + log
db.ErrorMode = 2  'PROD: Only MsgBox
db.ErrorMode = 2  'PROD: Log only msg in text file
```

- `demRaise (0)` : **log + Err.Raise**
  □ Ideal for development/testing: stops the macro, logs the error, helps you fix quickly.
- `demMsgBox (1)` : **MsgBox only**
  □ Demo / interactive usage (non-technical user).
  □ Note: this mode does **not** write to the log (current design).
- `demLogOnly (2)` : **log only** *(default)*
  □ Production: no popups, no interruptions, everything is traced.
  □ Use alongside an application strategy (e.g., check return values, show a user summary at the end).

By default, the log is written to `ThisWorkbook.Path & "\duckdb_errors.log"` (if `LogFilePath` is not set).

Each entry is appended with a timestamp:

```
2026-02-03 09:41:12 - QueryFast FAILED:
SELECT ...
DLL says: ...
```

When an operation fails:

1) The DLL returns `0`
2) `cDuck` calls `Native_LastErrorText()` (helper in `mDuckNative.bas`) to retrieve the native message
3) `cDuck.HandleError` builds a full message:
   - your context (e.g., “QueryFast FAILED…”, “OpenReadOnly FAILED…”)
   - – the DLL message (“DLL says: …”)
4) The message is stored in `cDuck.LastError` **and** handled according to `ErrorMode` (Raise / MsgBox / Log).

Main functions

DuckDB connections: “:memory:”, “.duckdb” file, and read-only

Open an in-memory database (`:memory:`)

```
db.OpenDuckDb ":memory:"
```

What is it for?

- fast pipelines, intermediate computations, temporary staging
- no file created, no disk I/O
- perfect for “load → transform → export”

Caution:

- everything disappears on close (or if the `cDuck` object is destroyed)

- if you need persistence: use a `.duckdb` file or export (Parquet/CSV).

Persistent database (`.duckdb`)

```
db.OpenDuckDb ThisWorkbook.Path & "\cache.duckdb"
```

Benefits:

- persistence (catalog + tables + metadata)
- portable database (single file)

Points to watch:

- locking/concurrency: avoid opening the same file for write from multiple processes.
- if multiple workbooks/processes are reading: prefer **read-only** mode.

Read-only (`OpenReadOnly`): safety and reporting mode

```
db.OpenReadOnly ThisWorkbook.Path & "\cache.duckdb"
```

This is ideal for:

- reporting / exploration
- protecting the database from accidental writes
- multi-process read usage (more predictable)

Any write command will fail (and will be logged/handled according to `ErrorMode`).

Lifecycle & automatic cleanup

- `CloseDuckDb` cleanly closes the connection.
- `Class_Terminate` (destructor) performs best-effort cleanup:
  - rollback if a transaction is open
  - close the connection if needed

Convenience singleton (optional)

`mDuckNative.bas` provides “single instance” access:

- `CurrentDuckDb` : creates a global instance if needed
- `CloseCurrentDuckDb` : closes and releases it

It is convenient for a monolithic Excel app, but remember the singleton: - centralizes state (handle, transaction, etc.) - must be properly closed at the end of the session.

Executing SQL and managing transactions

```
db.Exec "CREATE TABLE t(a INT, b TEXT);"
db.Exec "INSERT INTO t VALUES (1,'x');"
```

`Exec` : - normalizes SQL (trim + removes trailing `;` via `TrimSQL`) - logs an error if the handle is null (DB not open) - on failure: calls `HandleError` with a query excerpt

```
Public Sub QuickSmokeTest()
    Dim db As New cDuck
    Dim a As Variant

    db.Init ThisWorkbook.Path
    db.ErrorMode = 0              '0=Err.Raise, 1=MsgBox, 2=LogOnly
    db.OpenDuckDb ":memory:"
```

```
    db.Exec "CREATE TABLE t(a INT, b TEXT);"
    db.Exec "INSERT INTO t VALUES (1,'x'),(2,'y');"

    a = db.QueryFast("SELECT * FROM t ORDER BY a;")
    Call ArrayToSheet(a, Sheet1, "A1")  'helper mDuckNative.bas

    db.CloseDuckDb
End Sub
```

For write batches:

```
db.BeginTx
db.Exec "INSERT INTO ..."
db.Exec "UPDATE ..."
db.Commit
```

QueryFast(selectSql)

`QueryFast` executes a `SELECT` and returns a paste-ready `Variant(2D)`:

```
Dim v As Variant
v = db.QueryFast("SELECT * FROM my_table;")
Call ArrayToSheet(v, Sheet1, "A1")
```

Default choice: direct, fast, clean.

`SelectToSheet` goes through a CSV and then `QueryTables.Add`. It is slower than `QueryFast`, but can be useful if you want to rely on Excel's import pipeline (encoding, parsing, etc.).

Recommendation: use `QueryFast` for analytics; use `SelectToSheet` only if you have an Excel-specific constraint.

Scalar: a single value (COUNT/MAX/…) without fetching a table

```
Dim n As Long
n = db.ScalarLong("SELECT COUNT(*) FROM t;")
```

Scalar is perfect for:

- counters, bounds, flags
- checks before export
- presence tests

Demo: `Mod2DuckDb_Scalar.bas`

FrameFromValue

Creates a DuckDB "frame" table from a `Variant(2D)` (often `Range.Value2`):

```
Dim a As Variant
a = Sheet1.Range("A1").CurrentRegion.Value2
db.FrameFromValue "__frame", a, True, True   'temp table
```

Use case: instant staging to immediately do joins/aggregations in SQL.

AppendArray(tableName, data2D, hasHeader)

Feeds a DuckDB table via the **Appender** (very fast), without CSV/ADO:

```
db.Exec "CREATE TABLE stage(...);"
db.AppendArray "stage", a, True
```

"Dataframe-like" demo: `Mod2DuckDb_PythonLike.bas` (`Test_FrameFromValue_PythonLike`)

Special functions

UpsertFromArray(tableName, data2D, headerRow, keyColsCsv)

Use Excel as a data-entry/edit UI to update/insert data into a DuckDB database using one or more worksheet keys:

- **DB → Excel**: `ReloadFromDuckToExcel` (reload a table to a sheet)
- **Excel → DB**: `PushExcelToDuck` (push an Excel range and perform an **UPSERT**)

Conditions:

1) The target DuckDB table exists (`tableName`)

2) The sheet contains a tabular block starting at `A1` :
   - **row 1 = headers**
   - following rows = data
3) Excel headers are the **exact** DuckDB column names (same spelling).
4) Key columns must be **present** in the Excel range and in the table.
5) Keys must be **non-null** (no empty key).
6) Ideally, the key (or key combination) is **unique** in Excel *and* in the DB.

Excel → DB: `PushExcelToDuck`

1) Opens `demo.duckdb` read/write

2) Reads the Excel block from `A1` via:
   - `arr = ws.Range("A1").CurrentRegion.Value`
3) (Debug) prints:
   - DB columns (`information_schema.columns`)
   - Excel headers (row 1 of the array)
4) Calls:
   - `db.UpsertFromArray tableName, arr, 1, keyCols`
5) Re-reads the table and refreshes the sheet (visual check)

```
Dim a As Variant
a = Sheet1.Range("A1").CurrentRegion.Value2
db.UpsertFromArray "main.prices", a, 1, "ISIN,Date"
```

- `headerRow=1` : row 1 contains column names.
- `keyColsCsv` : key list (case-insensitive, normalized).

Use case: reference data updates, historization, incremental sync.

Demo: `Mod2DuckDb_ExcelUpdate.bas` (`PushExcelToDuck` , `ReloadFromDuckToExcel`)

How UPSERT really works (internal mechanics)

1) **Map Excel columns → DuckDB columns**
   - if `headerRow=1` : mapping by **header names**
2) **BEGIN TRANSACTION** (atomicity + performance)
3) **Create a TEMP staging table**

- e.g., `temp.__tmp_upsert`
- with the required columns in the correct order

4) **Ultra-fast ingestion** of Excel rows into the temp table (Appender)
5) Generic SQL UPSERT (without MERGE):
   - **UPDATE … FROM temp** on keys
   - **INSERT … WHERE NOT EXISTS** for new rows
6) Cleanup: `DROP temp.__tmp_upsert`
7) **COMMIT** (or ROLLBACK on error)

Large filters: Temp List to replace WHERE IN (…)

To avoid an enormous `WHERE IN (...)`.

1) `CreateTempList "__basket", keys, "TEXT"` → creates `temp.__basket(v TEXT)` and fills it quickly
2) SQL: `... WHERE isin IN (SELECT v FROM __basket)` **or** `JOIN __basket ON ...`

Option A: "building block" (`CreateTempList` + free SQL)

```
db.CreateTempList "__keys", keys1D, "VARCHAR"
v = db.QueryFast("SELECT * FROM t JOIN __keys k ON t.ISIN = k.v;")
```

Option B: "turn-key" (`SelectWithTempList`)

```
v = db.SelectWithTempList("__keys", keys1D, "VARCHAR", "main.Instruments", "ISIN", True)
```

Demos: - `Mod2DuckDb_WhereInSimple.bas` - `Mod2DuckDb_WhereInFct.bas`

Dictionary functions

Fills a `Scripting.Dictionary` from DuckDB data. The query must return a column used as the key.

- **DictFlat**: `key → value` (1 column)
- **DictRow1D**: `key → Variant(1D)` (multiple columns, *without* labels)
- **DictRow2D**: `key → Variant(2D)` (multiple columns, *with* labels)

When multiple rows share the same key:

Parameters:

- `onDupMode = 0`: **ignore** duplicates → keep the **first** occurrence
- `onDupMode = 1`: **replace** → keep the **last** occurrence encountered
- `clearFirst=True`: `dict.RemoveAll` before filling
- `clearFirst=False`: merge (useful if you build the cache in multiple passes)

A) key → value

- `SelectToDictFlat(selectSql, keyCol, [valCol]) As Dictionary`
- or `FillDictFlat(..., dict, ...)`

```
Dim db As cDuck, d As Dictionary, k As Variant
Set db = CurrentDuckDb
db.OpenDuckDb ThisWorkbook.Path & "\market.duckdb"
```

```
Set d = db.SelectToDictFlat( _
    "SELECT isin, name FROM securities;", _
    "isin", "name", True, 1)

For Each k In d.Keys
    Debug.Print k, d(k)
Next
```

Demo: `Mod2DuckDb_DictFlat.bas` (e.g., `EX_ISIN_ToName_Dict`, `Test_DictFlat_Duplicates`)

B) Row1D: key → 1D array of values

`SelectToDictRow1D(selectSql, keyCol, dict, [valueColsCsv])`
Ideal if you want multiple columns with index-based access.

Demo: `Mod2DuckDb_DictRow1D.bas`

```
Dim d As New Dictionary, vals As Variant

db.SelectToDictRow1D _
    "SELECT * FROM T", _
    "k", d, _
    "Name,ISIN", True, 1

vals = d("B")
Debug.Print vals(1), vals(2)    ' (1)=Name, (2)=ISIN
```

C) Row2D: key → small 2×N table (headers + values)

`SelectToDictRow2D(selectSql, keyCol, dict, ...)`
More readable (headers included), slightly heavier, enables searching a value by its column label.

For each key: - `dict(key)` is a **2D array** (bounds 1..2 × 1..(ncol-1)): - row 1: column names (excluding the key) - row 2: corresponding values (typed)

Demo: `Mod2DuckDb_DictRow2D.bas`

Data import

1) **Direct scan/query** (no persistent table)

```
SELECT ... FROM read_parquet('data.parquet');
```

2) **CTAS** (*Create Table As Select*): materialize a table

```
CREATE TABLE t AS SELECT * FROM read_csv_auto('data.csv');
```

3) **COPY FROM**: load into an existing table (often the most “controlled”)

```
COPY t FROM 'data.csv' (AUTO_DETECT TRUE);
```

A) Auto read: ReadToArray(filePath, [tailSql])

- **CSV**: native support (no extension).
- **Parquet**: often available by default, but the toolkit also provides `db.TryLoadExt "parquet"` / `db.LoadExt "parquet"` (see `Mod1DuckDb_Extension.bas`).
- **JSON**: supported via the `json` extension (often auto-loaded, but you can force it: `db.TryLoadExt "json"` / `db.LoadExt "json"`).

```
v = db.ReadToArray("C:\data\prices.parquet", "WHERE date >= DATE '2025-01-01'")
ArrayToSheet v, Sheet1, "A1"
```

The wrapper chooses the right function based on the file extension (Parquet / JSON / CSV auto).

B) CSV

- Import: `ImportCsvReplace(table, csvPath)` (create/replace)
- Export: `SelectToCsv(selectSql, csvPath)` or via `COPY`

Demos: `Mod1DuckDb_Csv.bas` (`Demo_ImportCsv`, `Demo_CSV_AutoDetect`, `Demo_CsvJoin`)

`read_csv_auto`
Auto-detects delimiter/quote/header/types, with robustness options.

```
SELECT *
FROM read_csv_auto('data.csv');
```

Common options

```
SELECT *
FROM read_csv_auto(
  'data.csv',
  delim=';',
  header=true,
  sample_size=-1,          -- read more to infer schema
  ignore_errors=true,      -- skip "broken" lines
  nullstr=['', 'NULL']     -- values to interpret as NULL
);
```

`read_csv` (explicit schema / strong control)
Useful when you **know** types or when auto-detection is unstable.

```
SELECT *
FROM read_csv(
  'data.csv',
  columns={'id':'BIGINT', 'dt':'TIMESTAMP', 'amount':'DOUBLE'},
  delim=';',
  header=true
);
```

Clean import into a table

```
CREATE TABLE main.sales AS
SELECT * FROM read_csv_auto('sales.csv');
```

`COPY ... FROM` (append / controlled load)

```
CREATE TABLE main.sales (...); -- schema already defined
COPY main.sales FROM 'sales.csv' (AUTO_DETECT TRUE);
```

C) Parquet (recommended for large volumes)

- export: `CopyToParquet(selectSql, outParquet)`
- read: `read_parquet(...)`

Demos: - `Mod1DuckDb_Parquet.bas` - `Mod1DuckDb_Parquet_Research.bas`

`read_parquet` **(standard)**

```
SELECT * FROM read_parquet('data.parquet');
```

Replacement scan (DuckDB infers via extension)

```
SELECT * FROM 'data.parquet';
```

Import table (CTAS)

```
CREATE TABLE main.fact AS
SELECT * FROM read_parquet('fact_*.parquet', union_by_name=true);
```

Read multiple files / datasets

- glob: `read_parquet('dataset/**/*.parquet')`
- list: `read_parquet(['a.parquet','b.parquet','c.parquet'])`
- Hive-style partitions:

```
SELECT * FROM read_parquet('orders/*/*/*.parquet', hive_partitioning=true);
```

D) JSON: NDJSON or JSON array

- `cDuck.CopyToJson(selectSql, outJson, overwrite:=True, boolJsonArray:=False)`
  - `boolJsonArray=False` → **NDJSON** (one object per line)
  - `boolJsonArray=True` → **JSON Array** (`[ {...},{...} ]`)
- `cDuck.CopyToJsonx(selectSql, outJson, overwrite:=True)`
  - forces `FORMAT JSON` (NDJSON by default).

Demos: `Mod1DuckDb_Json.bas`

`read_json_auto` **(your first reflex)**

DuckDB often figures it out by itself (records / format / nesting).

```
SELECT * FROM read_json_auto('data.json');
```

Control the format** (ndjson vs array vs unstructured)**

```
SELECT * FROM read_json_auto('data.ndjson', format='newline_delimited');
SELECT * FROM read_json_auto('data.json',   format='array');
```

Control object "unpacking" (`records` **)**

- `records=true`: turns JSON keys into columns.
- `records=false`: keeps JSON in a `STRUCT/JSON` column.

```
SELECT * FROM read_json_auto('data.ndjson', records=true);
```

NDJSON (default)

```
COPY main.events TO 'events.ndjson' (FORMAT json);
```

JSON Array (often requested by APIs)

```
COPY main.events TO 'events.json' (FORMAT json, ARRAY true);
```

Export a SELECT

```
COPY (
  SELECT id, created_at, payload
  FROM main.events
  WHERE created_at >= date '2026-01-01'
) TO 'events_2026.ndjson' (FORMAT json);
```

E) Access database

MS Access → DuckDB (4 methods)

1) **ODBC** `odbc_query` : Access executes the query; DuckDB retrieves the result
2) **ODBC** `odbc_scan` : raw table copy from Access into DuckDB
3) **ADO Recordset** → `AppendAdoRecordsetFast` : fast ingestion via the bridge DLL
4) **ADO Recordset** → **Variant(2D)** → `AppendArray` : universal fallback, more RAM

Demos: - `Mod3DuckDb_1AccessToDuck_main.bas` : `TestMain_AccessToDuckDB` - `Mod3DuckDb_2AccessToDuck_dll.bas` : normal vs fast benchmark - `mDuckNative.bas` : `CopyAccessToDuck_ODBC` , `AccessTable_ToParquet` , …

Method	Who does the "heavy lifting"?	SQL to write	Strengths	Limits / risks	Prerequisites
ODBC / `odbc_query`	**Access/ACE** interprets the query; DuckDB retrieves the result	**Access/ACE SQL** (with `[]` , Access functions, JOIN…)	• Very flexible • Often **very fast** if filters/joins run in Access	• More "fragile" (Access SQL dialect)• Driver errors possible• Not DuckDB SQL	• DuckDB `odbc` / `nanodbc` extension• + **Access ODBC driver**
ODBC / `odbc_scan`	**DuckDB** (via ODBC extension)	DuckDB SQL around it (`WHERE` , `LIMIT` …)	• Simple, robust for **raw table copy**	• Less flexible on Access side• Filters/joins mainly in DuckDB	• DuckDB `odbc` / `nanodbc` extension• + **Access ODBC driver**
ADO Recordset → `AppendAdoRecordset` *(bridge DLL)*	**Bridge DLL** reads the Recordset + inserts into DuckDB	Access SQL (in `rs.Open`)+ DuckDB (target table)	• No DuckDB ODBC extension required• Good performance• No huge VBA array	• Depends on ACE OLEDB• Cursor/forward-only settings matter	• **Microsoft.ACE.OLEDB.12.0**• + `duckdb.dll` • + `duckdb_vba_bridge.dll`
ADO Recordset → Variant(2D) → `AppendArray` *(bridge DLL)*	**VBA** materializes Variant(2D), then the **bridge DLL** ingests via `AppendArray`	Access SQL (in `rs.Open`)+ DuckDB (target table)	• Practical fallback• You can inspect/display the array• Final ingestion still runs in the **DLL**	• More **RAM** (everything in VBA memory)• Often slower on large volumes	• ACE OLEDB• + `RecordsetToVariant2D` • + `duckdb_vba_bridge.dll` (for `AppendArray`)

Data export

CSV export

A) Export a table

```
COPY main.sales TO 'sales.csv' (FORMAT csv, HEADER true);
```

B) Export a SELECT

```
COPY (
  SELECT client_id, sum(amount) AS total
```

```
  FROM main.sales
  GROUP BY 1
) TO 'sales_agg.csv' (FORMAT csv, HEADER true);
```

C) Useful options (locale / Excel)

```
COPY main.sales TO 'sales_fr.csv' (
  FORMAT csv,
  HEADER true,
  DELIMITER ';',
  QUOTE '"',
  ESCAPE '"',
  NULLSTR ''
);
```

Parquet export

Simple export

```
COPY main.fact TO 'fact.parquet' (FORMAT parquet);
```

Performance / format options

```
COPY main.fact TO 'fact_zstd.parquet' (
  FORMAT parquet,
  COMPRESSION zstd,
  ROW_GROUP_SIZE 250000
);
```

Partitioned write (lakehouse folder)

```
COPY main.fact
TO 'fact_dataset'
(FORMAT parquet, PARTITION_BY (year, month));
```

Metadata & maintenance (catalog, renames, checks)

Main functions (`cDuck`):

- `TablesInfo([schema])` : list tables/views
- `ColumnsInfo(table)` : columns/types
- `TableExists("schema.table")` / `ColumnExists(...)`
- `RenameTable` , `RenameColumn`
- `SelectShape` : SELECT dimensions without fetching data

Demo: `Mod2DuckDb_Info.bas` (`Test_TableExists` , `Test_ColumnExists` , `Test_RenameTableColumn` , `Demo_PRAGMA_CheatSheet`)

DuckDB extensions

DuckDB was designed as an **embedded** (*in-process*) and **extensible** engine: a significant share of features is not hard-coded in the core, but delivered as **extensions** that can be loaded on demand. An extension is a **binary** that DuckDB loads dynamically to enable new formats, functions, scanners, etc.

- Overview (docs): https://duckdb.org/docs/stable/extensions/overview
- Community extensions portal: https://duckdb.org/community_extensions/

Two families: *Core extensions* vs *Community extensions*

- **Core extensions**: “official” extensions, distributed and tested with DuckDB releases (support level varies by extension).
- **Community extensions**: extensions developed/maintained by the community (quality varies, but many are excellent).

DuckDB caches extension binaries in a user directory, typically under:

`%USERPROFILE%\.duckdb\extensions\<version>\windo`

Upgrading DuckDB (`v1.4.3` → `v1.4.4`) changes the version subfolder.

From DuckDB VBA:

- `INSTALL <ext>;` downloads/installs the extension into the local cache.
- `LOAD <ext>;` loads the extension into the current session (activates functions).
- Either run these commands (`db.Exec "INSTALL …"; db.Exec "LOAD …"`)
- Or use your wrapper `db.LoadExt "<ext>"` (recommended), which encapsulates best effort.

When DuckDB is updated, the cached extension may no longer match (ABI/signature/version). The simplest fix:

```
FORCE INSTALL rapidfuzz FROM community;
LOAD rapidfuzz;
```

Core (official) extensions

- Maintained by the DuckDB team.
- Distributed via the official repository.
- Typical examples: `parquet` , `json` , `httpfs` , `ui` (depending on distribution/docs).

“Community” extensions

- Contributed by third parties (not maintained by DuckDB Labs).
- Installable via the community extensions repository.
- Examples in your modules: `miniplot` , `rapidfuzz` , `nanodbc`

miniplot `miniplot` adds SQL functions that generate **charts** (often HTML/JS) directly from DuckDB.

From Excel/VBA: - you generate a chart in SQL - you retrieve either an **HTML file generated** (if supported by the function) - or an **HTML fragment returned** as SQL output, which you write yourself into a file.

- `bar_chart(labels_list, values_list, title [, output_path])`
- `line_chart(x_list, y_list, title [, output_path])`
- `scatter_chart(x_list, y_list, title [, output_path])`
- `area_chart(x_list, y_list, title [, output_path])`
- `scatter_3d_chart(x_list, y_list, z_list [, label_list], title [, output_path])`

RapidFuzz RapidFuzz adds high-performance SQL functions (C++ library) for:

- edit distance
- similarity
- variants (Jaro-Winkler, prefix/postfix, partial match…)

Then score only the remaining candidates:

```
SELECT *,
       rapidfuzz_ratio(lower(name), lower(?)) AS score
FROM t
WHERE score >= 70
ORDER BY score DESC
LIMIT 50;
```

- `rapidfuzz_ratio(a,b)` : general similarity (score)
- `rapidfuzz_partial_ratio(a,b)` : substring match (useful for tolerant “contains”)
- `rapidfuzz_jaro_winkler_*` : excellent for proper names / short typos
- `rapidfuzz_prefix_*` / `rapidfuzz_postfix_*` : prefix/suffix comparison
- `rapidfuzz_osa_*` : OSA distance (handles adjacent transpositions)

nanoODBC DuckDB can load an ODBC extension that allows it to:

- connect to any source with an ODBC driver
- read tables / execute queries
- materialize results inside DuckDB

Two modes: `odbc_scan` vs `odbc_query`

odbc_scan (raw copy)

- You provide a `table_name`
- DuckDB reads it “as-is”
- Great for staging / migrating full tables

odbc_query (push query to the source)

- You provide SQL **interpreted by Access/ACE** via ODBC
- Very attractive if you want to filter/join in Access before transfer

Simple heuristic:

- **scan** = “raw copy”
- **query** = “smart extraction” (pushdown)
- `LoadExt(name)` : loads (INSTALL + LOAD at DLL level)
- `TryLoadExt(name) As Boolean`
- `EnsureOdbcLoaded()` : tries `odbc` , otherwise `nanodbc`

Demos: - `Mod1DuckDb_Extension.bas` - `Mod3DuckDb_3Nanodbc_Ext.bas` - `Mod2DuckDb_RapidFuzz_Ext.bas` - `Mod2DuckDb_Miniplot_Ext.bas`

UI extension: local web interface

The `ui` extension starts a small local HTTP server and opens a UI in the browser.

Ideal for: - inspecting catalogs, tables, queries - prototyping SQL on a `.duckdb` database - “data engineering” debugging without leaving Excel

Your technical approach is solid:

- a global `gUiDuck` to keep the connection alive
- open **read-only** (`OpenReadOnly`) to be safe

- `CALL start_ui();` to start
- `CALL stop_ui_server();` + close to stop

5. Code explanation

Technical notes (DLL)

- **Interface:** UTF-16 on the VBA side, UTF-8 conversion on the DLL side
- **Strings:** all `...W` functions consume UTF-16 strings passed through `StrPtr(...)`.
- **Errors:** return 0 = failure. Human-readable details via `Duck_LastErrorW` (UTF-16 buffer).
  - Most functions return `int`: **1 = success**, **0 = failure**; on failure, a message is stored in a thread-local error buffer.
- **Identifier normalization:** SQL is pre-processed—identifiers `[...]` and `` `...` `` are converted to `"..."`, while string literals `'...'` and `"..."` remain intact.
- **Paths:** backslashes `\` are normalized to `/` inside the DLL (useful for `COPY TO`).
- **Cleanup:** all DuckDB resources (`result`, `prepare`, `appender`) and COM resources (`SAFEARRAY`, `VariantClear`) are systematically released.
- **Date/Time**
  - OLE Automation date (`VT_DATE`) via `SystemTimeToVariantTime` / `VariantTimeToSystemTime`.
  - Time-only mapped to `VT_DATE` relative to 1899-12-30 (Excel convention).
- DuckDB appender used for fast ingestion (faster than `INSERT`).
- Automatic `BEGIN/COMMIT` transactions in UPSERT and appender functions.
- **Unicode / wide strings:** `...W` and `StrPtr`; all `W`-suffixed functions expect **UTF-16** (wide) strings. In VBA, you pass a pointer to the string: `StrPtr(s)` → `LongPtr`.

SAFEARRAY: - `SelectFill2D_TypedV` expects a pre-dimensioned `Variant(2D of Variant)` (row 1 = headers). - `QueryToArrayV` allocates and fills a `Variant(2D)` (row 1 = headers). - Arrays are column-major.

- **Typing (SELECT outputs):**
  - Integers → **VT_R8** (Excel-friendly), floats → **VT_R8**.
  - `DATE/TIME/TIMESTAMP` → **VT_DATE** (OLE base).
  - `VARCHAR/TEXT/DECIMAL/UUID/INTERVAL` → **VT_BSTR**.
  - `BLOB` → `SAFEARRAY(Byte)` 1D (`VT_ARRAY|VT_UI1`).
- **Appender:** ultra-fast **RAM** ingestion (via `AppendArrayV`) without CSV or ADO.
- **Extensions:** `DuckVba_LoadExtW` performs **INSTALL** + **LOAD** (ignores “already installed” INSTALL errors).

Open/close & execute SQL

```
void* __stdcall DuckVba_OpenW(const wchar_t* db_path);
void* __stdcall DuckVba_OpenReadOnlyW(const wchar_t* db_path);
int   __stdcall DuckVba_Close(void* handle);
int   __stdcall DuckVba_ExecW(void* handle, const wchar_t* sql);
```

DuckVba_OpenW

```
Public Declare PtrSafe Function DuckVba_OpenW Lib "duckdb_vba_bridge.dll" (ByVal pwszPath
↪  As LongPtr) As LongPtr
```

- Purpose: open a DuckDB database read/write or read-only (a `.duckdb` file or `":memory:"`) and return a connection handle.
- Input: `dbPath` pointer to the path string.
- Output: non-zero `LongPtr` = handle; 0 on failure (read `Duck_LastErrorW`).

- Usage: `h = DuckVba_OpenW(StrPtr("C:...\cache.duckdb"))`

Arguments - `pwszPath`: pointer to `UTF-16` (wide). Examples: - `":memory:"` → in-RAM database - `"C:\...\my.duckdb"` → file database - UNC paths possible (`\\server\share\...`) but beware locks/latency.

Cautions - File DB: may be locked by another process. - Do not forget `DuckVba_Close`. - Do not share a handle across threads (VBA is not thread-safe).

DuckVba_OpenReadOnlyW

Opens a database in read-only mode.

```
Public Declare PtrSafe Function DuckVba_OpenReadOnlyW Lib "duckdb_vba_bridge.dll" (ByVal
↪  pwszPath As LongPtr) As LongPtr
```

Arguments - `pwszPath`: wide DB path.

DuckVba_Close

```
Public Declare PtrSafe Function DuckVba_Close Lib "duckdb_vba_bridge.dll" (ByVal h As
↪  LongPtr) As Long
```

- Purpose: cleanly closes the connection associated with the handle and releases resources.
- Returns: 1 on success, 0 otherwise.

Arguments - `h`: session handle.

DuckVba_ExecW

```
DuckVba_ExecW(h As LongPtr, sql As LongPtr) As Long
```

- Purpose: executes arbitrary SQL (DDL/DML, COPY, etc.).
- Returns: 1 on success, 0 on failure (details via `Duck_LastErrorW`).
- Notes: the bridge automatically rewrites identifiers `[...]` and `` `...` `` into double quotes `"..."` to match DuckDB.

Arguments - `h`: open session - `pwszSql`: wide SQL. - multi-statements? often yes, but prefer one statement per `Exec`. - trailing `;` usually optional.

Use cases - DDL: `CREATE TABLE`, `DROP`, `ALTER` - DML: `INSERT`, `UPDATE`, `DELETE` - control: `BEGIN`, `COMMIT`, `ROLLBACK` - export: `COPY (...) TO ...`

DuckVba_QueryToArrayFastV

Executes a SELECT and fills a `Variant(2D)`

```
Public Declare PtrSafe Function DuckVba_QueryToArrayFastV Lib "duckdb_vba_bridge.dll"
↪  (ByVal h As LongPtr, ByVal pwszSelect As LongPtr, ByRef v As Variant) As Long
```

Arguments - `h`: session - `pwszSelect`: wide SELECT - `v`: out parameter; receives a 2D array.

Duck_LastErrorW

```
Public Declare PtrSafe Function Duck_LastErrorW Lib "duckdb_vba_bridge.dll" (ByVal pwszBuf
↪  As LongPtr, ByVal cch As Long) As Long
```

- Purpose: get the last error message (native UTF-8 → UTF-16 in buffer).
- Returns: number of characters copied (excluding NUL). 0 if none.

Arguments - `pwszBuf` : pre-allocated wide buffer on the VBA side. - `cch` : capacity in characters.

DuckVba_FrameFromValue

Creates a DuckDB "frame" table from a `Variant(2D)` .

```
Public Declare PtrSafe Function DuckVba_FrameFromValue Lib "duckdb_vba_bridge.dll" (ByVal h
↪  As LongPtr, ByVal pwszFrame As LongPtr, ByRef v As Variant, ByVal hasHeader As Long,
↪  ByVal makeTemp As Long) As Long
```

Arguments

- `pwszFrame` : table name (temp or persistent depending on `makeTemp`).
- `v` : 2D array.
- `hasHeader` :
  - `1` : first row = column names.
  - `0` : columns auto-named (likely `col1` , `col2` , …).
- `makeTemp` :
  - `1` : temporary table (session scope)
  - `0` : normal (persistent) table—depending on open mode

DuckVba_AppendArrayV

```
Public Declare PtrSafe Function DuckVba_AppendArrayV Lib "duckdb_vba_bridge.dll" (ByVal h
↪  As LongPtr, ByVal pTable As LongPtr, ByRef v As Variant, ByVal hasHeader As Long) As
↪  Long
```

- Purpose: feeds a DuckDB table directly from a `Variant(2D)` (e.g., `Range.Value2`) without CSV.
  - Parameters:
    - `v` : 2D array; if `hasHeader=1` , the first row is skipped (treated as headers).
    - Type mapping: BOOL → bool, numbers → DOUBLE/INT, DATE → TIMESTAMP, text → VARCHAR, Byte() → BLOB.
- When: very fast in-RAM bulk push thanks to the appender.

Use cases - load an Excel extract into DuckDB to run joins/aggs - stage a `__tmp__` table for upsert - etc.

DuckVba_ScalarV

Executes a SELECT returning a single value (1 row, 1 col) and outputs that scalar.

```
Public Declare PtrSafe Function DuckVba_ScalarV Lib "duckdb_vba_bridge.dll" (ByVal h As
↪  LongPtr, ByVal pwszSelect As LongPtr, ByRef v As Variant) As Long
```

Arguments - `pwszSelect` : typically `SELECT COUNT(*) FROM ...`

Return - non-zero = OK, `0` = FAIL - `v` : value (Variant), may be `Null/Empty` if result is empty.

DuckVba_LoadExtW

Loads a DuckDB extension.

```
Public Declare PtrSafe Function DuckVba_LoadExtW Lib "duckdb_vba_bridge.dll" (ByVal h As
↪  LongPtr, ByVal pName As LongPtr) As Long
```

Arguments - `pName` : extension name (wide). Examples: - `"parquet"` - `"json"` - `"odbc"` / `"nanodbc"`

Use cases - enable `read_parquet` , `parquet_schema` , `read_json_auto` - enable `odbc_scan` , `odbc_query` for Access/ODBC

Caution - extension files must be in the expected DuckDB cache location.

Info

DuckVba_TableInfoV

Returns catalog metadata on tables/views.

```
Public Declare PtrSafe Function DuckVba_TableInfoV Lib "duckdb_vba_bridge.dll" (ByVal h As
↪  LongPtr, ByVal pwszSchemaFilter As LongPtr, ByRef v As Variant) As Long
```

Arguments - `pwszSchemaFilter` : - `0` /NULL → all schemas - otherwise `"main"` , `"temp"` , etc.

DuckVba_ColumnsInfoV

Returns metadata about columns of a table.

```
Public Declare PtrSafe Function DuckVba_ColumnsInfoV Lib "duckdb_vba_bridge.dll" (ByVal h
↪  As LongPtr, ByVal pwszTable As LongPtr, ByRef v As Variant) As Long
```

DuckVba_TableExistsW

Checks whether a table/view exists.

```
Public Declare PtrSafe Function DuckVba_TableExistsW Lib "duckdb_vba_bridge.dll" (ByVal h
↪  As LongPtr, ByVal table_path_w As LongPtr) As Long
```

Arguments - `table_path_w` : `"main.clients"` , `"temp.__tmp__"` , `"clients"`

Return - non-zero = exists, `0` = does not exist (or error).

DuckVba_ColumnExistsW

Checks whether a column exists in a table.

```
Public Declare PtrSafe Function DuckVba_ColumnExistsW Lib "duckdb_vba_bridge.dll" (ByVal h
↪  As LongPtr, ByVal pwszTablePath As LongPtr, ByVal pwszColName As LongPtr) As Long
```

Arguments - `pwszTablePath` : table - `pwszColName` : column

Return - non-zero = exists, `0` = does not exist (or error)

DuckVba_RenameTableW

Renames a table.

```
Public Declare PtrSafe Function DuckVba_RenameTableW Lib "duckdb_vba_bridge.dll" (ByVal h
↪  As LongPtr, ByVal pwszOldTable As LongPtr, ByVal pwszNewTable As LongPtr) As Long
```

Arguments - `pwszOldTable` : may include schema - `pwszNewTable` : same

Return - non-zero = OK, `0` = FAIL

DuckVba_RenameColumnW

Renames a column.

Prototype

```
Public Declare PtrSafe Function DuckVba_RenameColumnW Lib "duckdb_vba_bridge.dll" (ByVal h
↪  As LongPtr, ByVal pwszTable As LongPtr, ByVal pwszOldCol As LongPtr, ByVal pwszNewCol
↪  As LongPtr) As Long
```

Arguments - `pwszTable` : table - `pwszOldCol` / `pwszNewCol` : columns

Return - non-zero = OK, `0` = FAIL

SELECT → Excel / SAFEARRAY

DuckVba_SelectToCsvW

```
Public Declare PtrSafe Function DuckVba_SelectToCsvW Lib "duckdb_vba_bridge.dll" (ByVal h
↪  As LongPtr, ByVal pwszSelect As LongPtr, ByVal pwszCsv As LongPtr) As Long
```

- Purpose: executes a SELECT and writes a CSV directly (UTF-8, header).
- Return: 1 on success.
- When: load later via `QueryTables` without going through ADO.

Arguments - `pwszSelect` : wide SELECT - `pwszCsv` : wide CSV path

DuckVba_SelectShapeW

Returns result dimensions (without returning data).

```
Public Declare PtrSafe Function DuckVba_SelectShapeW Lib "duckdb_vba_bridge.dll" (ByVal h
↪  As LongPtr, ByVal pwszSelect As LongPtr, ByRef outRows As Long, ByRef outCols As Long)
↪  As Long
```

DuckVba_SelectFill2D_TypedV (Private)

“Typed” variant of SELECT→Variant, aiming to better preserve types (Date/Number) instead of converting everything to text.

Use cases - when Excel must recognize dates/numbers without parsing. - intermediate exports where typing matters (dates, timestamps).

DuckVba_ReadCsvToTableW

```
DuckVba_ReadCsvToTableW(handle, table, csv_path, create_if_missing)
```

- If the table does not exist:
  - `create_if_missing = 1` → `CREATE TABLE AS SELECT * FROM read_csv_auto(...)`
  - `create_if_missing = 0` → error “table not found”
- If the table exists: append via `COPY table FROM 'file' (AUTO_DETECT=TRUE, HEADER=TRUE)`

DuckVba_ExecPreparedToArrayV

Executes a prepared statement and returns a `Variant(2D)`.

```
Public Declare PtrSafe Function DuckVba_ExecPreparedToArrayV Lib "duckdb_vba_bridge.dll"
↪  (ByVal ps As LongPtr, ByRef v As Variant) As Long
```

Arguments - `ps`: prepared handle (returned by `DuckVba_PrepareW`) - `v`: out result

Return - non-zero = OK, `0` = FAIL

DuckVba_PrepareW

Prepares a parameterized statement (placeholders `?`).

```
Public Declare PtrSafe Function DuckVba_PrepareW Lib "duckdb_vba_bridge.dll" (ByVal h As
↪  LongPtr, ByVal pwszSql As LongPtr) As LongPtr
```

Arguments - `h`: session - `pwszSql`: wide SQL, typically: - `"INSERT INTO t VALUES (?, ?, ?);"` - `"SELECT * FROM t WHERE id=?;"`

Return - `ps` prepared handle, or `0` if FAIL.

Use cases - prevent injection: do not concatenate user values - stabilize the plan: better performance in loops

DuckVba_ExecPrepared

Executes the prepared statement (no table returned).

```
Public Declare PtrSafe Function DuckVba_ExecPrepared Lib "duckdb_vba_bridge.dll" (ByVal ps
↪  As LongPtr) As Long
```

Use cases - parameterized INSERT/UPDATE - parameterized DDL (less common)

DuckVba_Finalize

Releases the prepared statement.

```
Public Declare PtrSafe Function DuckVba_Finalize Lib "duckdb_vba_bridge.dll" (ByVal ps As
↪  LongPtr) As Long
```

DuckVba_CopyToParquetW

Exports a SELECT to Parquet.

```
Public Declare PtrSafe Function DuckVba_CopyToParquetW Lib "duckdb_vba_bridge.dll" (ByVal h
 ↪  As LongPtr, ByVal pSel As LongPtr, ByVal pOut As LongPtr) As Long
```

Arguments - `pSel`: wide SELECT - `pOut`: wide parquet path

Return - non-zero = OK, `0` = FAIL.

Use cases - produce a "lakehouse" dataset from Excel/Access - archive compressed snapshots - share large data without Excel

Special functions

DuckVba_AppendAdoRecordset

Purpose: derives schema from ADO Fields and reads a table (MS Access via COM) → optional CREATE TABLE, and **ingests** into DuckDB via Appender.

```
Public Declare PtrSafe Function DuckVba_AppendAdoRecordset Lib "duckdb_vba_bridge.dll"
 ↪  (ByVal h As LongPtr, ByVal pRecordset As LongPtr, ByVal pwszTable As LongPtr, ByVal
 ↪  createIfMissing As Long) As Long
```

Arguments - `pRecordset`: `ObjPtr(rs)` - `pwszTable`: target table - `createIfMissing`: - `1`: create table if missing (types derived from ADO schema) - `0`: table must exist

COM techniques - Resolves `Fields`, `Fields.Count`, `Fields.Item(i)`, then on each Field: `Name`, `Type`, `Value`.
- **Infers** DuckDB schema (`ado_type_to_duck`):
`adBoolean→BOOLEAN, adBigInt→BIGINT, Single/Double→DOUBLE, Currency→DECIMAL(19,4), Date/DateTime→DATE/TIMES`
- `create_if_missing=1` → `DROP TABLE IF EXISTS` + `CREATE TABLE "name"(…)` (safe quoting).
- Row loop: `MoveFirst` best-effort then `EOF` / `MoveNext`. ADO→DuckDB appender mapping analogous to `AppendArrayV`.

DuckVba_AppendAdoRecordsetFast

Optimized variant for large volumes.

Technique - bulk fetch (`GetRows`) or forward-only streaming, fewer COM round-trips.

DuckVba_UpsertFromArrayV

Upsert (merge) from a `Variant(2D)` into a target table using one or more keys.

```
Public Declare PtrSafe Function DuckVba_UpsertFromArrayV Lib "duckdb_vba_bridge.dll"
 ↪  (ByVal h As LongPtr, ByVal pwszTable As LongPtr, ByRef v As Variant, ByVal headerRow As
 ↪  Long, ByVal pwszKeyColsCsv As LongPtr) As Long
```

Purpose - Updates or inserts rows from an Excel array (maps columns by header names at `header_row`, otherwise by position) into a DuckDB table. - UPSERT **without MERGE** (version-compatible and generic SQL). - Runs an `UPDATE … FROM` then an `INSERT … WHERE NOT EXISTS` inside `BEGIN … COMMIT`, with temp table create/drop. - Key columns must be present in the mapped columns (otherwise the join cannot be performed).

Arguments - `pwszTable`: target table - `v`: 2D array containing at least the required columns (including keys) - `headerRow`: header row index in `v` - usually `1` if row 1 contains column names - `pwszKeyColsCsv`: CSV list of key columns (wide), e.g.: - `"ISIN"` - `"ISIN,ContractNumber"`

Exact pipeline 1) **Discover target columns**
- `SELECT column_name FROM information_schema.columns WHERE schema/table`.
- Build `tgt_cols[]` + `tgt_cols_norm[]` (via `normalize_ident_w`) for case-insensitive comparisons.

2) **Parse keys**
   - `key_columns_csv_w` → segments (`,`), `trim_ws_dup` + `normalize_ident_w`.
   - Validate presence in the selection.
3) **Map Excel columns → table**
   - If `header_row>=1` and in bounds → read headers (BSTR) and normalize, **match by name**.
   - Complement with **positional** mapping without collision (e.g., if some columns are unnamed).
   - Build `sel_tgt_idx[]` and `sel_xls_idx[]` (only used columns).
4) **BEGIN TRANSACTION** (atomicity + performance).
5) **CREATE TEMP TABLE __tmp_upsert AS SELECT FROM WHERE 0;**
   - `<qtab>` = `"schema"."table"` (safe quoting).
   - Temp table has **the exact structure** of the selection.
6) **Fill TEMP via Appender**
   - Iterate useful rows (`rstart = header_row+1` if headers).
   - COM→DuckDB mapping identical to `AppendArrayV`.
7) **UPDATE…FROM + INSERT missing**
   - `is_key[i]` identifies join columns.
   - **UPDATE**: `UPDATE <qtab> SET non_keys = s.non_keys FROM temp s WHERE <qtab.key = s.key AND …>`
   - **INSERT**: `INSERT INTO <qtab>(cols) SELECT s.cols FROM temp s WHERE NOT EXISTS(SELECT 1 FROM <qtab>`
   - Build fragments: `set_list`, `where_update`, `where_exists`, `ins_cols`, `ins_vals` (same quoting).
8) **DROP TEMP + COMMIT**
   - `DROP TABLE IF EXISTS temp.__tmp_upsert` (best-effort).
   - COMMIT, otherwise ROLLBACK on error.

DuckVba_CreateTempListV

Purpose: creates a TEMP table `tabname(v <sqlType>)` and fills it from a Variant (1D or 2D → first column).

```
Public Declare PtrSafe Function DuckVba_CreateTempListV Lib "duckdb_vba_bridge.dll" (ByVal
↪  h As LongPtr, ByVal tabname_w As LongPtr, ByRef keys As Variant, ByVal sqltype_w As
↪  LongPtr) As Long
```

- Types: `sqlType` is indicative (VARCHAR, INT, DOUBLE…); the DLL converts values.
- When: for `WHERE x IN (SELECT v FROM tabname)` or `JOIN tabname t ON ...`.
- Benefit: much more efficient for lists with thousands of IDs than a gigantic `IN (...)`.

Technique - Detects the target "kind" (`TMP_VARCHAR` / `TMP_INT64` / `TMP_DOUBLE`) via `sql_type` (simple keyword match).
- Prefers **Appender**; falls back to **prepared INSERT** if appender fails (schema "temp" unavailable → retry "main").

Arguments - `tabname_w` : temp table name (wide) - `keys` : - expected: list of values (Variant 1D, or 2D 1-col) - types: Strings/LongLong/Double depending on `sqltype_w` - `sqltype_w` : explicit SQL type (e.g. `"VARCHAR"` , `"BIGINT"` , `"DOUBLE"`) - forces the key type in DuckDB, avoids weak inference

DuckVba_SelectWithTempList2V

Turn-key API: creates/fills a temp list and returns a filtered/joined SELECT.

```
Public Declare PtrSafe Function DuckVba_SelectWithTempList2V Lib "duckdb_vba_bridge.dll"
 ↪  (ByVal h As LongPtr, ByVal pTabName As LongPtr, ByRef keys As Variant, ByVal pSqlType
 ↪  As LongPtr, ByVal pSelectOrTable As LongPtr, ByVal pJoinCol As LongPtr, ByVal autoJoin
 ↪  As Long, ByRef vOut As Variant) As Long
```

Arguments - `pTabName` : key table name - `keys` : key values (Variant) - `pSqlType` : DuckDB type of keys (wide) - `pSelectOrTable` : - either a **table name** (e.g. `"main.clients"`) - or a **SELECT query** (e.g. `"SELECT * FROM ..."`). - `pJoinCol` : - wide pointer to join column, or `0` /NULL if not provided. - `autoJoin` : - `1` : DLL tries to auto-determine the join (*inferred*: by common column name, first column, unique column, etc.) - `0` : explicit join required via `pJoinCol` . - `vOut` : result as `Variant(2D)`

Internal technique 1) create temp list (like `CreateTempListV`) 2) build SQL such as: - `SELECT ... FROM (<selectOrTable>)` 3) execute and convert to `Variant(2D)`

DuckVba_SelectToDictW

Fills a Dictionary where each key points to a **row** (often a 1D/2D array) depending on implementation.

```
Public Declare PtrSafe Function DuckVba_SelectToDictW Lib "duckdb_vba_bridge.dll" (ByVal h
 ↪  As LongPtr, ByVal pwszSelect As LongPtr, ByVal pwszKeyCol As LongPtr, ByVal pDict As
 ↪  LongPtr, ByVal clearFirst As Long, ByVal onDupMode As Long) As Long
```

- Purpose: executes a SELECT and fills a provided `Scripting.Dictionary` (via `ObjPtr(dict)`).
- Key: the `keyCol` column (case-insensitive, tolerates `[]` , `` ` `` and `"`).
- Value: a 2×J SAFEARRAY:
- Row 1 = headers (all columns except the key)
- Row 2 = values (typed: BOOL/DOUBLE/DATE/BSTR/BLOB)
  - Options:
    - `clearFirst=1` → `dict.RemoveAll` before.
    - `onDupMode=0` → ignore duplicates; `1` → replace (`Item(key) = val`).
- When: ideal for key→row access in VBA (e.g. `D("FR0002")`).

DuckVba_SelectToDictFlatW

Fills a flat Dictionary `key → value` **.**

```
Public Declare PtrSafe Function DuckVba_SelectToDictFlatW Lib "duckdb_vba_bridge.dll"
 ↪  (ByVal h As LongPtr, ByVal pwszSelect As LongPtr, ByVal pwszKeyCol As LongPtr, ByVal
 ↪  pwszValCol As LongPtr, ByVal pDict As LongPtr, ByVal clearFirst As Long, ByVal
 ↪  onDupMode As Long) As Long
```

Arguments - `pwszValCol` : - may be NULL (0) for “auto” if the SELECT returns exactly 2 columns (**observed in your class** via `pVal=0&`). - otherwise: value column name.

DuckVba_SelectToDictValsColsW

Fills a Dictionary key → “values from selected columns”.

```
Public Declare PtrSafe Function DuckVba_SelectToDictValsColsW Lib "duckdb_vba_bridge.dll"
 ↪  (ByVal h As LongPtr, ByVal pwszSelect As LongPtr, ByVal pwszKeyCol As LongPtr, ByVal
 ↪  pwszValueColsCsv As LongPtr, ByVal pDict As LongPtr, ByVal clearFirst As Long, ByVal
 ↪  onDupMode As Long) As Long
```

Arguments - `pwszValueColsCsv` : - `0` /NULL → “all columns except the key” (suggested by your wrapper) - otherwise `"colA,colB,colC"` .

Appender & ingestion

```
void* DuckVba_AppenderOpen(void* h, const wchar_t* schema, const wchar_t* table);
int   DuckVba_AppenderClose(void* app);
int   DuckVba_AppenderBeginRow(void* app);
int   DuckVba_AppenderEndRow(void* app);
int   DuckVba_AppendNull(void* app);
int   DuckVba_AppendBool(void* app, int b);
int   DuckVba_AppendInt64(void* app, long long v);
int   DuckVba_AppendDouble(void* app, double v);
int   DuckVba_AppendVarcharW(void* app, const wchar_t* w);
int   DuckVba_AppendDateYMD(void* app, int y, int m, int d);
int   DuckVba_AppendTimestampYMDHMSms(void* app, int y,int m,int d,int hh,int mm,int
 ↪  ss,int ms);
int   DuckVba_AppendBlob(void* app, const void* data, long long len);
int   DuckVba_AppendArrayV(void* h, const wchar_t* table, VARIANT* pvar, int has_header);
```

Appendix

Parquet file

A Parquet file is a column-oriented format optimized to store and process large volumes of data.

Parquet is a **columnar** and **binary** storage format introduced in 2013 (initiated notably by Cloudera and Twitter), designed for large-scale analytics, especially in Hadoop/Spark environments and **data lake / lakehouse** architectures. Unlike CSV, which organizes data by **rows**, Parquet stores it by **columns**. This columnar layout allows query engines to read only the columns they need (instead of scanning the full table), while benefiting from much more efficient compression and encoding.

With Parquet, it is generally unnecessary to load an entire table into memory to analyze it: the engine reads only the blocks and columns required by the query, leveraging metadata and schema stored in the file. Data is read only where it is useful.

CSV limitations

Classic “flat” formats like CSV emphasize simplicity and portability, but have major limitations for analytics:

- Text storage (costly parsing, type ambiguities),
- Row-oriented organization, which forces reading a lot of irrelevant data when only a few columns are needed.
- No compression: larger files increase storage and transfer costs.
- Limited optimization opportunities (little or no exploitable metadata).

Parquet addresses these limitations with a binary, columnar format designed for query engines.

Column format (Parquet)

Parquet stores values column by column, **grouped into blocks** (row groups / column chunks) and accompanied by metadata (schema, statistics, encoding, compression). Result: engines can limit reading to only the required columns (**column pruning**) and sometimes skip entire blocks (predicate pushdown), which dramatically speeds up analytical processing.

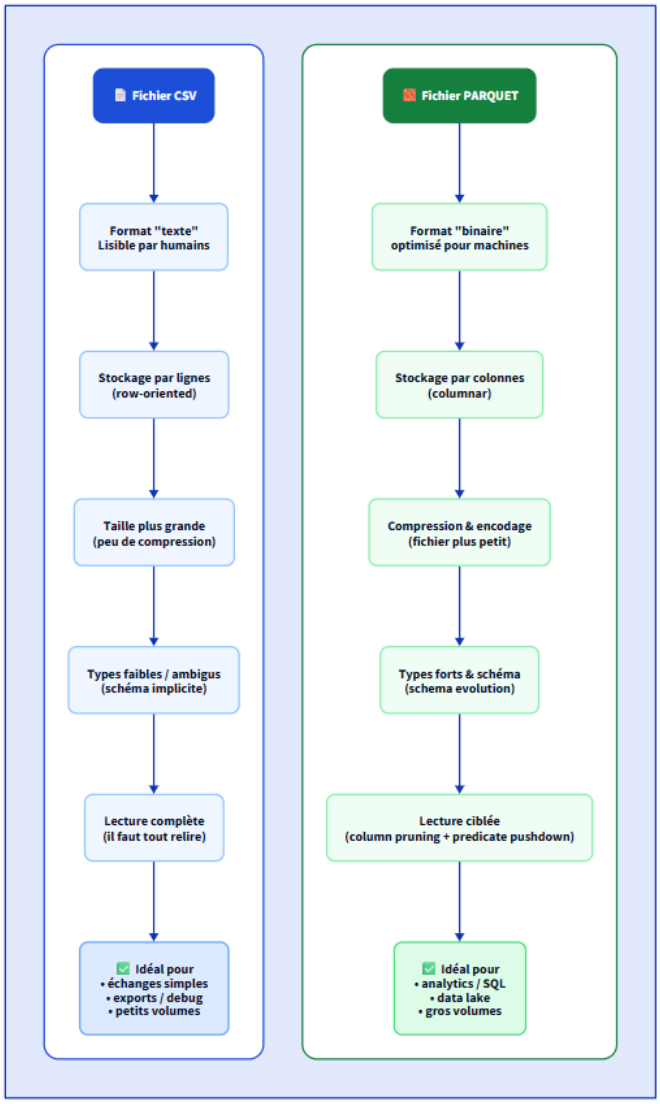
- **Space savings**: a 1 TB CSV dataset reduced to ~130 GB in Parquet (87% savings)
- **Speed gains**: a SQL query going from 236 s (CSV) to 6.78 s (Parquet), ~×34 faster, depending on engine and query type.

Internal structure (technical principles)

Parquet files are divided into row groups, each containing a batch of rows. Each row group is divided into column chunks, each containing data for a column. These chunks are further divided into smaller units called pages, which are compressed to save space. Parquet files also contain additional footer information called metadata, which helps locate and read only the data we need.

- **Data typing (schema)**: Parquet embeds the schema (types: int, float, bool, date…). Unlike CSV (all text), this reduces conversions, ambiguities (dates, separators) and typing errors.
- **Compression**: Parquet supports multiple codecs (Snappy, Gzip, Zstd…). Because values within a column are often similar, compression is usually very effective, which reduces disk usage and the amount of data read.
- **Encoding (data encoding)**: Before or in addition to compression, Parquet applies columnar encodings (e.g., Dictionary Encoding for repeated values, RLE, bit-packing). Result: more compact data and faster decoding.
- **Parallel read/write**: Parquet is split into internal blocks (row groups/pages), enabling distributed engines (e.g., Spark) to read and process in parallel—accelerating large-volume processing.

- **Rich metadata (query optimization)**: Parquet stores metadata and statistics per block (e.g., min/max, number of nulls). SQL engines can use them to avoid reading irrelevant blocks when filtering (predicate pushdown), reducing scans and improving performance.



OLTP vs OLAP

OLTP

OLTP stands for **Online Transaction Processing**. It is a database designed to run an application day-to-day. It must handle many small, fast operations (read/update a record, create an order, validate a payment) with strong integrity guarantees (ACID transactions) and **minimal latency**. It is optimized to access a small number of rows through indexes, and most often uses **row-oriented** storage (row-store), well-suited to targeted reads/writes.

Technical details

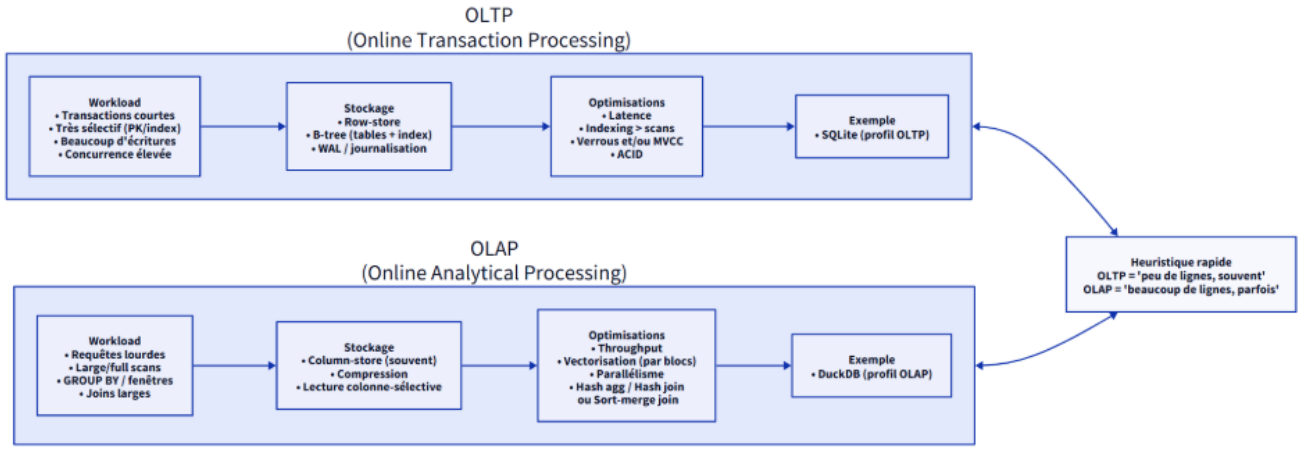
The storage model is typically row-store with **B-tree** structures (data + indexes), because you often read/write multiple columns of the same row. Key mechanisms include **ACID**, concurrency control (locks, MVCC depending on the DBMS), **write-ahead logging (WAL)**, and query plans that favor indexing over scanning. Schemas are often normalized to reduce redundancy and maintain consistency during updates.

OLAP

OLAP stands for **Online Analytical Processing**. It is a database designed to **analyze and aggregate data**. It executes less frequent but much heavier queries (scans, GROUP BY, windows, wide joins) over **large volumes** to produce indicators, reports, or ad hoc exploration. It is optimized to read many rows while touching only certain columns, which favors **column-oriented** storage (column-store), with **compression**, **vectorized** execution, and often **parallelism** to maximize throughput.

Technical details

It is an engine optimized for **CPU/memory-intensive** queries over **large volumes**, with throughput (MB/s → GB/s) as the primary goal rather than per-query latency. Typical workloads involve full/large scans, aggregations, wide joins, window functions, and selective column reads. Hence the frequent use of **column-store**, which enables **compression**, better **cache locality**, and vectorized execution (block processing). Aggregations also benefit from algorithms such as hash aggregation and joins like **hash join** or **sort-merge join** depending on distributions. Data is often organized into **denormalized** schemas or **star** schemas (facts/dimensions) to reduce join cost and speed up aggregations.



OLTP: MySQL, PostgreSQL, MariaDB, SQLite, IBM Db2, Oracle Database, Microsoft SQL Server.

OLAP: DuckDB, ClickHouse, Google BigQuery, Snowflake, Amazon Redshift.

SQLite vs DuckDB

Criterion	Transactional (OLTP)	Analytical (OLAP)
Goal	Real-time unit transactions	Large-scale analytics on historical data
Storage model	Row-based	Column-based
Query execution	Iterator-based (row by row)	Vectorized (batch)
Data model	Highly normalized (3NF, strong integrity constraints)	Denormalized (star/snowflake, materialized views)
Access pattern	Random access (read/write few rows)	Sequential access (read millions of rows, few writes)
Optimized queries	Fast CRUD (`INSERT` , `UPDATE` , `DELETE` , simple `SELECT`)	Heavy aggregations, multiple joins, analytic functions
Storage structure	Row-store	Column-store (often compressed)
Indexing	B-Tree indexes, hash, full-text	Bitmap indexes, dictionaries, vectorized compression
Transactions	Strict ACID, low latency, high isolation	Fewer concurrent transactions, more batch reads
Concurrency	High (fine-grained locking, MVCC, many short threads)	Medium (CPU parallelism, vectorized execution)
Writes	Frequent, small atomic operations	Rare, batch loading (ETL, append-only)
Reads	Selective, by primary key or index	Massive sequential scans, optimized
Target performance	Minimal latency (ms)	Max throughput (MB/s → GB/s)
Caching	Page cache / buffer pool	Column block cache, compression, CPU vectorization
Typical dataset size	MB to a few GB	GB to TB
Operation frequency	Very high (thousands ops/s)	Low (heavy queries but less frequent)
History	“Live” short-lived data	Historical, often read-only
Query plan optimization	Index access and fast transactions	Column scans and parallel aggregations
Typical technologies	MySQL, PostgreSQL, SQLite, SQL Server, Oracle	DuckDB, ClickHouse, Snowflake, BigQuery, Redshift, Druid
Use cases	ERP, e-commerce, booking systems, mobile apps	Data warehouse, BI, reporting, exploratory analytics
Typical user	Real-time app, end-users	Analysts, data engineers, data scientists
Typical architecture	Distributed OLTP, master/slave replication	Data warehouse (warehouse/lakehouse)
Result materialization	Temporary results, isolated transactions	Stored aggregates, OLAP cubes, persisted results

MS Access vs SQLite vs DuckDB

Criterion	Microsoft Access	SQLite	DuckDB
Database type	Relational with UI (OLTP)	Embedded SQL engine (OLTP)	Embedded analytical engine (OLAP)
Philosophy	Office database, end-user oriented	Minimalist DB integrated into apps	Local analytics engine for data science
Main file	`.accdb` / `.mdb`	`.sqlite` / `.db`	`.duckdb`
Architecture	Row-by-row	Row-based	Columnar
Platforms	Mainly Windows	Windows, macOS, Linux, Android, iOS	Windows, macOS, Linux
Server required	No (local file)	No (serverless)	No (serverless)
User interface	Full GUI (forms, reports, VBA macros)	None (CLI or via code)	None (CLI, Python, R, SQL)
Languages / API	VBA, ODBC, .NET	C, Python, Go, Java, etc.	Python, R, C++, Java, SQL
Optimized queries	Simple CRUD, forms, reports	Small transactional queries	Analytical queries (aggregations, heavy joins)
Performance (small)	Good	Excellent	Very good
Performance (large)	Poor (slows > 1 GB)	Medium (a few GB)	Excellent (tens of GB)
Concurrent read/write	Very limited	Single writer	Multi-threaded, vectorized
Multi-user	Possible on LAN, often unreliable	Multi-read, single write	Efficient parallel reads
Indexing	Yes (B-Tree, primary, secondary)	Yes (B-Tree)	Often unnecessary (vectorized optimization)
ACID	Yes	Yes	Yes
SQL standard support	Partial (SQL + Access functions)	Good (SQL92-ish)	Very good (modern SQL + analytics)
Complex joins	Limited	Yes, basic	Yes, optimized for big volumes
Data types	Restricted, Access-oriented	Flexible typing	Rich typing (Arrow/Parquet compatible)
Excel compatibility	Excellent	Possible via ODBC	Excellent (direct read of `.csv`, `.xlsx`, `.parquet`)
Import / Export	CSV, Excel, SQL via UI	SQL, CSV via scripts	CSV, Parquet, Arrow, JSON, Excel
Automation	Macros, VBA	Language APIs	Python/R APIs or SQL
Max DB size	~2 GB	Up to multiple TB (FS-dependent)	Tens of GB (compressed, columnar)
Compression	No	No	Yes (columnar compression)
Record format	Microsoft proprietary	Lightweight open binary	Optimized columnar binary
Notebook usage	No	Yes (with modules)	Yes (native integration)

Criterion	Microsoft Access	SQLite	DuckDB
Read external files without import	No	Partial (extensions)	Yes (CSV, Parquet, Arrow, JSON)
Typical use	Internal management, small desktop apps	Local storage in web/mobile apps	Large file analytics, local ETL, data science
Examples	Inventory, simple billing, company forms	Mobile apps, websites, IoT	Dataset exploration, local ETL, analytics prototyping
Required skills	Low (office users)	Medium (development)	Medium to advanced (data/SQL)
License	Proprietary (Microsoft Office)	Open (Public Domain)	Open Source (MIT)
Community / support	Medium, office-oriented	Very large, developers	Fast-growing, data-oriented
File portability	Medium (Windows)	Excellent	Excellent
Security / encryption	Basic password	Extensions available	Basic (not fully featured yet)
Maintenance	Visual UI, less flexible	Manual via scripts	Manual or automated via code
Cloud / modern integration	Low	Medium (wrappers)	Good (data-lake formats)

LICENSE

DUCK VBA DLL — GNU General Public License v3.0

This project (**DUCK VBA DLL**, including the C/C++ bridge DLL, the VBA toolkit, sample files, and related documentation) is distributed under the **GNU General Public License, version 3** (**GPL-3.0-only**).

License notice (standard text)

This program is free software: you can redistribute it and/or modify it under the terms of the **GNU General Public License** as published by the **Free Software Foundation**, **version 3** of the license.

This program is distributed in the hope that it will be useful, but **WITHOUT ANY WARRANTY**; without even the implied warranty of **MERCHANTABILITY** or **FITNESS FOR A PARTICULAR PURPOSE**.
See the **GNU General Public License** for more details.

You should have received a copy of the **GNU General Public License** along with this program. If not, see: https://www.gnu.org/licenses/gpl-3.0.html

What it implies (practical summary)

- **Free use**, including in a commercial context.
- If you **redistribute** (source or binary), you must:
  - provide the **corresponding source code** (or an equivalent GPL-compliant offer),
  - keep **notices** (copyright, license),
  - distribute any modified version under **GPLv3** (copyleft).
- **No obligation** to publish your modifications if you do not redistribute (internal use).
- For **SaaS / network service** use, GPLv3 does not force code publication (unlike AGPL).

  Note: exact compliance depends on your distribution mode (workbooks, DLLs, package, installer, etc.).
  If in doubt, reread the GPLv3 terms and/or have it validated by your legal team.

Third-party components

This project interfaces with **DuckDB** (database engine), which is a third-party component under its own license (DuckDB is under **MIT**).
The GPLv3 above covers **DUCK VBA DLL**; you must also comply with the licenses of any third-party components you distribute (DuckDB, optional extensions, etc.).

Trademarks / Logo / Visual identity (Trademark Policy)

The names **“DUCK VBA DLL”**, **“Duck-VBA”** (if applicable), as well as the associated logos, icons, and visual identity elements (the **“Marks”**) are trademarks and/or distinctive signs owned by **Etienne Lenoir**.

The software is distributed under the **GNU General Public License v3.0 (GPLv3)**. **GPLv3 grants no rights to use the Marks.** Any use of the Marks must comply with trademark law and this policy.

Permitted uses (in general)

- **Nominative use** to describe or reference the software (documentation, comparisons, citations), provided it does not suggest endorsement, partnership, or affiliation.
- Redistribution of **unmodified copies** of the software while keeping the Marks **as provided**.

Prohibited uses without prior written permission

- Using the Marks in a way that **suggests endorsement**, sponsorship, or affiliation without authorization.
- Using the Marks as the main name/logo/branding of a **modified version**, fork, or derived distribution in a way that could create confusion with the official project.
- **Modifying, adapting, recoloring, redrawing, animating, distorting** or creating derivative versions of the logos/icons (unless explicitly agreed).

Practical rule for forks / modified versions

You may modify and redistribute the code under GPLv3, **but** you must **rename and/or rebrand** your distribution as necessary to avoid confusion with the official distribution.

Any goodwill generated by authorized use of the Marks benefits their owner.

References

1. DuckDB Contributors. DuckDB (official website). https://duckdb.org/
2. dbdb.io. DuckDB (Database of Databases). https://dbdb.io/db/duckdb
3. Kernighan, Brian W.; Ritchie, Dennis M. The C Programming Language (2nd Edition). Prentice Hall / Pearson, 1988. ISBN 978-0131103627. https://books.google.com/books/about/The_C_Programming_Language.html?id=FGkPBQAAQBAJ
4. Metz, Sandi; Owen, Katrina. 99 Bottles of OOP: A Practical Guide to Object-Oriented Design (2nd Edition). Potato Canyon Software, LLC, 2024. https://sandimetz.com/99bottles
5. Needham, Mark; Hunger, Michael; Simons, Michael. DuckDB in Action. Manning Publications, 2024. ISBN 978-1633437258.
6. Harbison, Samuel P.; Steele Jr., Guy L. C: A Reference Manual (5th Edition). Pearson, 2002. ISBN 978-0130895929.
7. Gustedt, Jens. Modern C. Manning Publications, 2019. ISBN 978-1617295812.
8. Seacord, Robert C. Effective C (2nd Edition): An Introduction to Professional C Programming. No Starch Press, 2024.