DUCK VBA DLL

Etienne Lenoir

February 5, 2026

Excel/VBA & Access: Upgraded with DuckDB

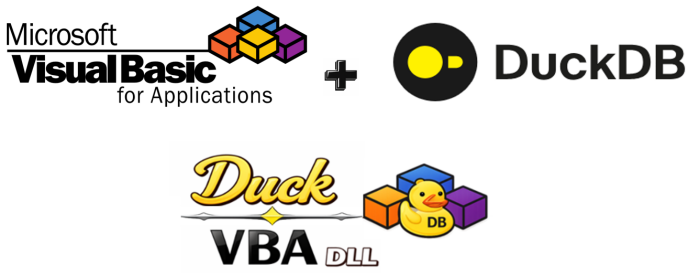
Duck VBA DLL — Kit DLL C/C++ & Toolkit VBA

Sommaire

1. Introduction

1.1 Présentation Duck VBA DLL

Duck-VBA-DLL intègre DuckDB à Excel/VBA : un moteur local ultra-performant, sans installation ni dépendances, et une base OLAP moderne pour le traitement analytique (OLAP), une alternative solide à Microsoft Access



Duck-VBA est une passerelle native (**DLL Windows en C/C++**) entre **VBA** (Excel/Office) et **DuckDB**, un moteur SQL analytique embarqué. L'objectif est simple : donner à VBA un **moteur de calcul ultra performant** et déployable facilement (un simple fichier .dll à côté du .xlsm).

Avec Duck-VBA, Excel reste le front-end (saisie, contrôles, reporting), tandis que DuckDB devient le back-end local pour tout ce qui coûte cher en temps et en complexité : JOIN, GROUP BY, fenêtres (WINDOW), CTE, upserts, staging, transactions, imports/exports (CSV/Parquet/JSON), et traitements volumineux.

Concrètement :

- Charge les données en base embarquée (depuis Excel, CSV, Parquet, JSON…)
- Traite et transforme en SQL côté DuckDB (moteur vectorisé, optimisé, multi-thread)
- Gére des millions de lignes

Le tout fonctionne soit en mode persistant (fichier .duckdb local, portable), soit en mode éphémère :memory: (en **RAM**) pour des pipelines ultra rapides sans I/O disque. Résultat : VBA digère enfin la donnée comme un outil data moderne, avec une **approche dataframe-like** (à la pandas) pilotée par SQL.

Le dll upgrade VBA en lui offrant une voie plus rapide, plus robuste et plus moderne… en déportant les opérations data vers le moteur SQL optimisé de DuckDB (vectorisé, multi-thread).

1.2 Contexte

VBA reste très utilisée en entreprise parce qu'elle est native à Office, déjà déployée partout, facile à partager via un simple .xlsm, et parfait pour le "dernier kilomètre" (reporting, saisie, automatisation, macros héritées). Même avec Python/BI, Excel reste l'interface universelle

Dès qu'on passe à la data "sérieuse" (volumes, ETL, SQL avancé, transactions), les approches Excel/VBA classiques (boucles, cellules, ADO/ODBC) deviennent vite lentes, fragiles et difficiles à industrialiser, et au-delà d'un certain seuil, les limites apparaissent vite :

- **Boucles VBA** : performances insuffisantes dès qu'on manipule des milliers/millions de lignes.
- **Power Query** : excellent pour du data prep, mais moins adapté aux workflows "macro-driven", parfois peu transparent, et limité sur les scénarios transactionnels (staging, upsert, synchronisation).
- **MS Access** : très pratique en "base locale", mais plafonne vite sur les usages data modernes (limite de taille de fichier,moins riche pour des transformations lourdes)

Duck-VBA répond précisément à ce point de rupture : il prolonge l'esprit "base locale type Access" en greffant à VBA un moteur SQL analytique embarqué (DuckDB), capable d'absorber de gros traitements (imports rapides, tables temporaires, requêtes complexes, upserts), tout en gardant Excel comme interface.

C'est la première intégration permettant à VBA de lire et d'écrire du **Parquet** directement (via DuckDB) sans dépendances.

1.3 Présentation de DuckDB

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

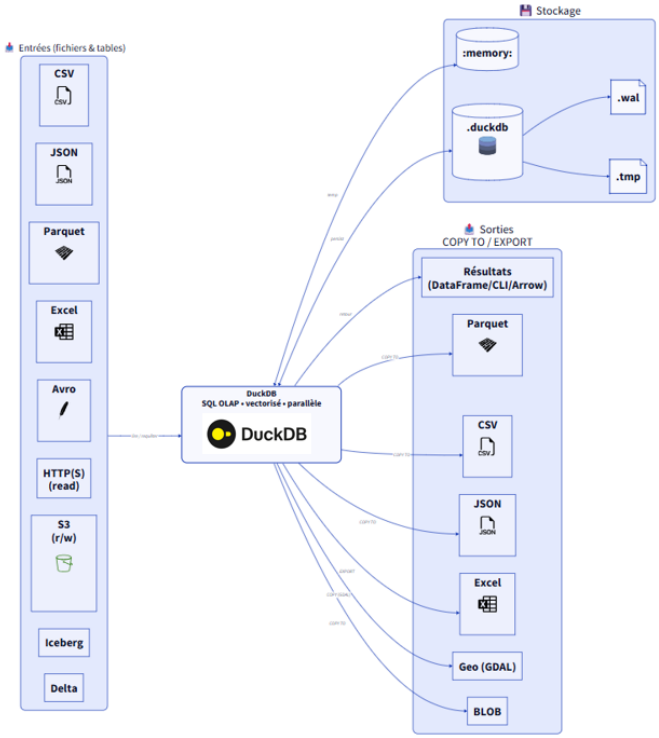
DuckDB est né au CWI[1] (Pays-Bas), initialement développé par **Mark Raasveldt** et **Hannes Mühleisen**. Le projet démarre vers **2018**, et une première version publique arrive en 2019,Le projet est publié sous licence MIT.

DuckDB est un moteur SQL analytique (**OLAP**) orienté **traitements colonnaires**, conçu pour maximiser le débit sur des volumes importants (scans, filtres, agrégations, jointures). Son exécution est vectorisée (traitement par lots) et parallélisée (multi-thread), ce qui lui permet d'exploiter efficacement le CPU et la mémoire sur des requêtes analytiques, en particulier lorsqu'on ne lit qu'un sous-ensemble de colonnes. Il est également très adapté aux formats colonnaires tels que **Parquet** (projection de colonnes, optimisation des lectures, pushdown).

DuckDB est **sans serveur (serverless)** : aucune instance à déployer, aucun service à maintenir, aucun port réseau à exposer. Il s'utilise soit en base **en mémoire** (`:memory:`) pour des traitements temporaires à faible latence, soit en base persistante via un fichier `.duckdb` (portable, local) pour stocker durablement les données, métadonnées et éventuels index.

Il ressemble à **SQLite** par son côté embarqué (zéro serveur, simple à intégrer, souvent un seul fichier), mais il s'en distingue par sa finalité : DuckDB est optimisé pour l'analytique (OLAP, gros scans et agrégations), tandis que SQLite est surtout optimisé pour le transactionnel (OLTP, petites opérations CRUD et transactions fréquentes).

Il lit et traite très bien les formats data modernes (CSV, Parquet, JSON,), permet des transformations SQL avancées, et s'intègre facilement dans un workflow local-first Il est conçu pour offrir des performances élevées sur les requêtes complexes interrogeant de grandes bases de données en configuration embarquée, notamment en combinant des tables comportant des centaines de colonnes et des milliards de lignes.



[1] Centrum voor Wiskunde en Informatica : Centre de recherche national en mathématiques et informatique à Amsterdam

1.4 Avantage Duck VBA DLL

L'objectif n'est pas de transformer VBA en langage data, mais de lui donner un **moteur SQL moderne** pour tout ce qui est coûteux en boucles Excel :

- joins, group by, fenêtrage, CTE, staging, transactions
- ingestion rapide (Appender) depuis arrays/recordsets
- formats modernes (Parquet/JSON) via extensions DuckDB
- retours "Excel-ready" : `Variant(2D)` (collage direct) ou `Dictionary` (lookups ultra rapides)

Avantages:

- ☒ Un vrai moteur de base de données moderne pour VBA : DuckDB embarqué in-process, sans serveur → SQL analytique rapide (JOIN, GROUP BY, WINDOW, CTE, transactions).
- ☒ Déploiement ultra simple : une DLL + un .xlsm, sans driver (pas d'ODBC/DSN), pas d'installation à gérer poste par poste.
- ☒ Alternative "base locale" à MS Access : fichier .duckdb portable (persistant) ou :memory: (100% RAM) pour pipelines temporaires sans I/O disque.
- ☒ Ingestion RAM/lecture data modernes : CSV / Parquet / JSON (et gros volumes) avec des scans et imports nettement plus efficaces que les approches VBA/ADO classiques
- ☒ Bridge Excel □ SQL optimisé :
  - Array / Range → DuckDB via appender (staging instantané, sans fichiers intermédiaires)
  - SELECT → Variant(2D) (collage direct sur feuille) ou → Dictionary (lookups quasi instantanés côté VBA)
- ☒ Workflow "dataframe-like" (pandas-inspired) : on part d'un array Excel en mémoire, on le matérialise en frame/table DuckDB (temp ou persistante), on applique les transformations en SQL (join/group/window/CTE/casts…), puis on renvoie le résultat en Variant(2D) (pour Excel) ou en Dictionary (pour des lookups ultra rapides).
- ☒ Mode :memory: (100% RAM) : pipelines très rapides sans I/O disque, parfait pour ETL jetable, calculs intermédiaires, tests, itérations — avec la puissance d'un moteur SQL vectorisé derrière tes arrays
- ☒ Fonctions "VBA-friendly" prêtes à l'emploi (spécifiques DLL) :
  - Upsert depuis Range/Array (sync Excel → DB : update + insert)
  - Exports dictionnaires (flat / row1D / row2D) pour mapping et accès rapide en mémoire
  - Temp lists pour remplacer les gros WHERE IN (…) (filtrage rapide sur milliers de clés)
  - Scalar helpers (COUNT/MAX/…) sans rapatrier une table complète
  - Passerelles Access → DuckDB : import de tables / recordsets, et export Access → Parquet / DuckDB selon les options (ODBC si dispo, sinon fallback ADO)

1.5 Qu'est ce qu'un DLL ?

Une DLL (Dynamic Link Library) est un module **binaire** natif Windows (format **PE – Portable Executable**), proche d'un .exe, mais conçu pour être **chargé par un autre programme** et lui fournir du code réutilisable. Il comprend un binaire PE contenant du code machine (opcodes) et des tables (imports/exports) qui décrivent comment le charger et le relier.

- **Link** = on connecte un programme à une bibliothèque
- **Dynamic** = la connexion se fait au moment où le programme tourne (pas au moment de la compilation)

Une DLL est chargée dès qu'une application en a besoin (au démarrage via ses imports, ou plus tard à la demande). Elle reste en mémoire tant qu'au moins un programme l'utilise : Windows maintient un compteur de références, et lorsque toutes les applications l'ont libérée, la DLL peut être **déchargée**

L'objectif principal des DLL est la **mutualisation** du code et des données :

- **Sur disque** : une seule copie de la bibliothèque est stockée.
- **En mémoire** : lorsque plusieurs processus utilisent la même DLL, une partie de ses pages peut être partagée, ce qui réduit la consommation globale de RAM (surtout pour le code et les données en lecture seule).

Le format d'un fichier DLL est identique à celui d'un exécutable (EXE). La principale différence réside dans le fait qu'une DLL ne peut être exécutée directement, car le système d'exploitation requiert un point d'entrée pour démarrer son exécution. Windows fournit un utilitaire (RUNDLL.EXE/ RUNDLL32.EXE) permettant d'exécuter une fonction exposée par une DLL.

Du code C à la DLL (ce qui se passe réellement)

1) Compilation (`cl.exe`)

Le compilateur MSVC prend le fichier source `.c` et produit un ou plusieurs fichiers objets `.obj` .

Ce `.obj` contient déjà du **code natif** (instructions x64 encodées en octets) ainsi que des informations nécessaires au lien :

- symboles (fonctions/variables)
- sections (code, données)
- informations de relocation
- références externes (imports à résoudre)

En pratique, la chaîne est : **préprocesseur → analyse/typage → optimisations (/O2) → génération de code**. Le résultat n'est plus du C : c'est un artefact binaire prêt à être assemblé/lié.

> NB : même si l'on ne génère pas explicitement un fichier `.asm` , il existe toujours une étape logique de *lowering* vers une représentation proche de l'assembleur, puis encodage en **opcodes** (code machine).

2) Édition de liens (`link.exe`)

Le linker construit `duckdb_vba_bridge.dll` à partir des `.obj` et des bibliothèques d'import :

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

Il produit une DLL au format **PE (Portable Executable)** et construit notamment :

- `.text` : code exécutable (opcodes x64)
- `.rdata` : constantes, tables en lecture seule
- `.data` : données modifiables
- `.idata` : *Import Directory* + **IAT (Import Address Table)** pour les dépendances
- `.edata` : *Export Directory* (fonctions exposées à VBA/Windows)
- `.reloc` : relocations (correctifs d'adresses si l'image est chargée à une autre base)

3) Résultat

La DLL finale est un **binaire natif** : elle contient exclusivement :

- du **code machine**

- des données
- des tables PE (imports/exports/relocations…)

Il n'existe plus aucune notion `C` à ce stade : uniquement des octets exploitables par le loader Windows et le CPU.

Comment Windows charge et exécute une DLL

Lorsqu'Excel/VBA charge la DLL (directement ou indirectement via `LoadLibrary`), le **loader Windows** effectue les opérations suivantes :

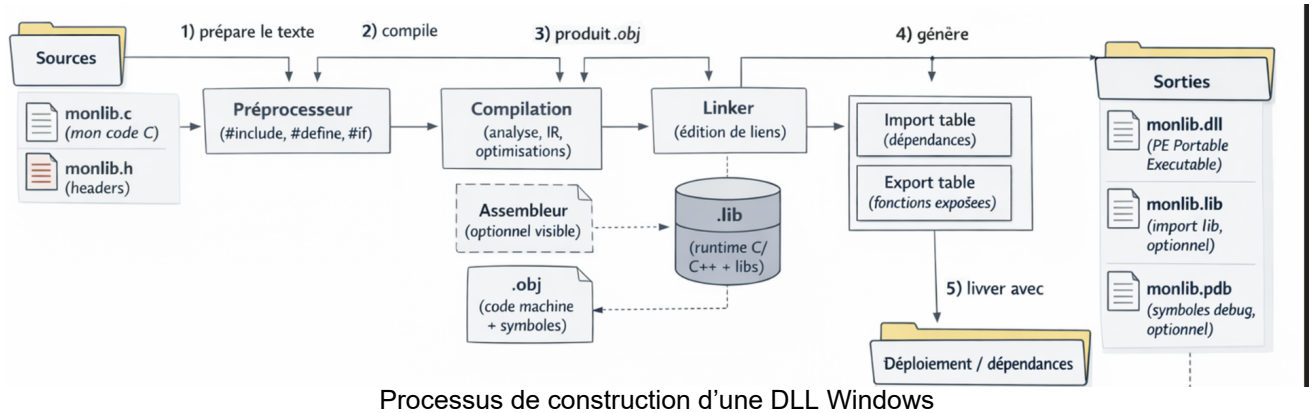
1) **Mapping en mémoire** des sections du PE (memory mapping)

2) Application des **protections mémoire** :
   - code en *Read + Execute*
   - données en *Read + Write*
3) Résolution des **imports** :
   - chargement des dépendances (`duckdb.dll` , `oleaut32.dll` , etc.)
   - remplissage de l'**IAT** avec les adresses réelles des fonctions importées
4) Application des **relocations** si la DLL n'est pas chargée à son *image base* préférée
5) Appel optionnel de `DllMain` (initialisation du module)

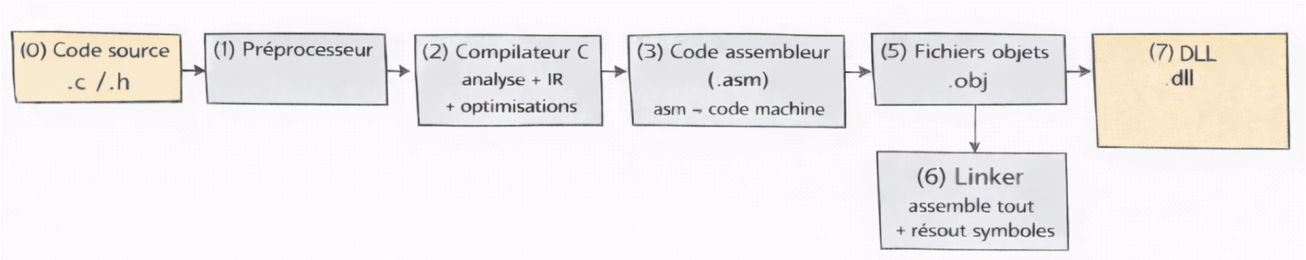
Ensuite, lorsqu'une fonction exportée est appelée depuis VBA :

- Excel récupère l'adresse de la fonction (Export Table / `GetProcAddress`)
- l'appel devient un simple transfert de contrôle vers une adresse mémoire dans `.text`
- le **CPU exécute directement** les opcodes x64 (*fetch → decode → execute*)

Autrement dit : Windows "comprend" une DLL via son format PE et ses tables ; le CPU exécute le code parce qu'il s'agit déjà d'instructions machine, pas de code interprété.



Processus de construction d'une DLL Windows



Pipeline simplifié : compilation DLL

2. Installation

Le dll `duckdb_vba_bridge.dll` est compilé et testé avec la `version DuckDB v1.4.3` (décembre 2025)

Prérequis :

- Windows
- Microsoft Visual C++ Redistributable (souvent déjà présent sur les postes entreprise/Office)
  - Le cas échéant il faudra les dll suivantes : vcruntime140.dll, msvcp140.dll, vcruntime140_1.dll dans le meme dossier.
- Excel 64 bits (VBA7)
- DuckDB runtime : `duckdb.dll` (binaire DuckDB officiel version 1.4.3)
- Bridge DUCK VBA DLL : `duckdb_vba_bridge.dll` (DLL project)

Déploiement (zéro installation) :

- Aucune installation système requise : pas de serveur, pas d'ODBC/DSN, pas de driver à installer.
- Portable : par défaut, les DLL peuvent être placées dans le même dossier que le .xlsm (ou un sous-dossier du projet).
- Chemin configurable : via la classe cDuck (ex. cDuck.Init), vous pouvez pointer vers un autre répertoire de binaires si besoin (poste verrouillé, dossier partagé, versioning, etc.).

Pour intégrer Duck-VBA dans un autre classeur/projet VBA, le minimum requis est d'importer le module `mDuckNative` et la classe `cDuck` (ils encapsulent l'API native et exposent l'interface principale)

Sécurité Windows : après copie/téléchargement, vérifiez que `duckdb.dll` et `duckdb_vba_bridge.dll` ne sont pas bloquées par Windows (clic droit Propriétés → cocher Débloquer → Appliquer), sinon Excel/VBA peut refuser de charger la DLL.

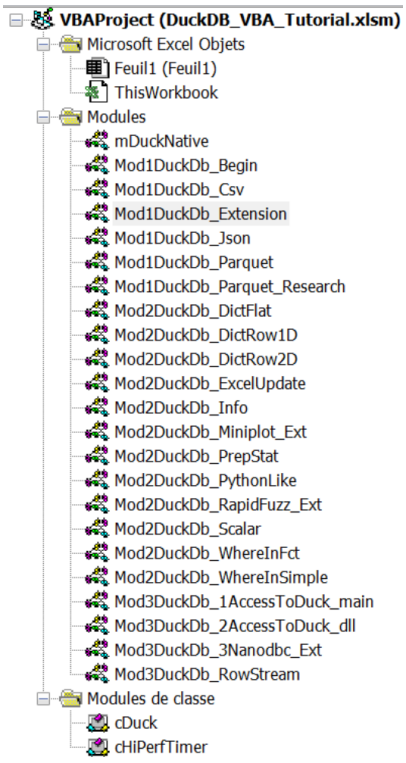
`Certaines fonctionnalités (ex. parquet, json, odbc/rapidfuzz) peuvent dépendre d'extensions DuckDB`

```
Dim db As New cDuck
'1) Init : indique à DuckVba où chercher les DLL (DuckVba.dll, duckdb.dll, etc.)
'   Ici : dossier du classeur (.xlsm). Classique si tu poses les DLL à côté du fichier.
db.Init ThisWorkbook.Path
'db.Init "C:\Tools\DuckVba"

'2)Ouvre DuckDB :
'":memory:" = base éphémère en RAM (disparaît à la fermeture, ultra rapide, zéro I/O
↪  disque)
db.OpenDuckDb ":memory:"

'Variante : base persistante sur disque (un seul fichier .duckdb portable)
db.OpenDuckDb ThisWorkbook.Path & "\cache.duckdb"
```

3. Tutoriel complet (Ficher démo)



Un tutoriel pas-à-pas est fourni dans le classeur `DuckDB_VBA_Tutorial.xlsm` Il contient des exemples prêts à exécuter (imports, requêtes, upserts, dictionnaires, parquet/json/csv, benchmarks…) et la documentation est directement dans les modules VBA (ouvre l'éditeur VBE : ALT + F11).

Recommandé : commencer par les modules `Begin / Info` puis suivre les démos (les procédures `Demo_*`).

Ou consulter la documentation de DuckDB qui est très complète :

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

Besoin	Méthode(s) `cDuck`	Démo (module)
Ouvrir/fermer DB	`Init`, `OpenDuckDb`, `OpenReadOnly`, `CloseDuckDb`	`Mod1DuckDb_Begin.bas`
Exécuter SQL	`Exec`, `ExecOK`	`Mod1DuckDb_Begin.bas`
SELECT → Excel	`QueryFast`, `SelectToSheet`	`Begin`, `Parquet`
Scalar (COUNT/…)	`Scalar*`	`Mod2DuckDb_Scalar.bas`
Charger array Excel	`FrameFromValue`, `AppendArray`	`Mod2DuckDb_PythonLike.bas`
Upsert Excel → DB	`UpsertFromArray`	`Mod2DuckDb_ExcelUpdate.bas`
Gros `WHERE IN`	`CreateTempList`, `SelectWithTempList`	`Mod2DuckDb_WhereIn*`
Lookup en mémoire	`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	Appender bas niveau	`Mod3DuckDb_RowStream.bas`

4. Démarrer avec Duck-VBA

Cette section documente **l'usage côté Excel/VBA** : comment ouvrir une connexion DuckDB, exécuter du SQL, charger/extraire des données, et choisir la bonne méthode selon le contexte (volumes, performance, robustesse, postes verrouillés…).

- `mDuckNative.bas` : *socle* (déclarations `Declare`, chargement des DLL, helpers pratiques).
- `cDuck.cls` : *façade* orientée usage (gestion d'erreurs, méthodes haut niveau, ergonomie), 1 instance de `cDuck` = 1 **connexion** DuckDB (handle).
- `db.Init(dllFolder)` appelle `EnsureDuckDll` (dans `mDuckNative.bas`) qui :
  - fixe le répertoire de recherche DLL (`SetDllDirectoryW`)
  - charge `duckdb.dll` + `duckdb_vba_bridge.dll` via `LoadLibraryW`

Placer `duckdb.dll` et `duckdb_vba_bridge.dll` **dans le même dossier** que le `.xlsm` ou dans le répertoire de votre choix

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

Fonctions Principales

- `OpenDuckDb(pathOrMemory)` : lecture/écriture
- `OpenReadOnly(path)` : lecture seule
- `CloseDuckDb`
- `Exec(sql)` : DDL/DML/COPY/PRAGMA…

Erreurs, logs & diagnostic

La DLL est “C-style” : la plupart des fonctions renvoient un statut (**1 = OK, 0 = KO**) et écrivent le détail dans un buffer d’erreur interne. Le wrapper VBA (`cDuck`) standardise ce comportement en une stratégie d’erreur claire et configurable.

```
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**
  □ Idéal en développement / tests : stoppe la macro, trace dans le log, permet de corriger vite.
- `demMsgBox (1)` : **MsgBox uniquement**
  □ Démo / usage interactif (utilisateur non-tech).
  □ À noter : ce mode **n’écrit pas** dans le log (par design actuel).
- `demLogOnly (2)` : **log uniquement** *(valeur par défaut)*
  □ Production : pas de popup, pas d’interruption, tout est tracé.
  □ À utiliser avec une stratégie applicative (ex. vérifier les retours, afficher un résumé utilisateur en fin de process).

Par défaut le message log est écrit dans `ThisWorkbook.Path & "\duckdb_errors.log"` (si `LogFilePath` n’est pas défini)

Chaque entrée est appendée avec timestamp :

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

Quand une opération échoue :

1) La DLL renvoie `0`
2) `cDuck` appelle `Native_LastErrorText()` (helper dans `mDuckNative.bas`) pour récupérer le message natif
3) `cDuck.HandleError` construit un message complet :
   - votre contexte (ex. “QueryFast KO…”, “OpenReadOnly KO…”)
   - – le message de la DLL (“DLL says: …”)
4) Le message est stocké dans `cDuck.LastError` **et** traité selon `ErrorMode` (Raise / MsgBox / Log).

Fonctions Principales

Connexions DuckDB :memory:, fichier .duckdb, et lecture seule

Ouvrir une base en mémoire (`:memory:`)

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

À quoi ça sert ?

- pipelines rapides, calculs intermédiaires, staging temporaire
- aucun fichier créé, pas d’I/O disque

- parfait pour “charger → transformer → exporter”

Attention

- tout disparaît à la fermeture (ou si l’objet `cDuck` est détruit)
- si vous avez besoin de persistance : utilisez un fichier `.duckdb` ou exportez (Parquet/CSV).

Base persistante (`.duckdb`)

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

Avantages

- persistance (catalogue + tables + métadonnées)
- base portable (un fichier)

Points d’attention

- verrouillage/accès concurrent : évitez d’ouvrir le même fichier en écriture depuis plusieurs processus.
- si vous avez plusieurs classeurs/processus en consultation : privilégiez la **lecture seule**.

Lecture seule (`OpenReadOnly`) : sécurité et mode reporting

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

Ce mode est idéal pour :

- reporting / exploration
- sécuriser la base contre les écritures accidentelles
- usage multi-process en lecture (plus prédictible)

Toute commande d’écriture échouera (et sera loggée/traitée selon `ErrorMode`).

Cycle de vie & nettoyage automatique

- `CloseDuckDb` ferme proprement la connexion.
- `Class_Terminate` (destructor) fait un **best-effort** :
  - rollback si une transaction est ouverte
  - fermeture de la connexion si nécessaire

Singleton pratique (optionnel)

`mDuckNative.bas` propose un accès “instance unique” :

- `CurrentDuckDb` : crée une instance globale si nécessaire
- `CloseCurrentDuckDb` : ferme et libère

C’est pratique pour une appli Excel “monolithique”, mais gardez en tête que le singleton : - centralise l’état (handle, transaction, etc.) - doit être fermé proprement en fin de session.

Exécuter du SQL et gérer les transactions

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

`Exec` : - normalise le SQL (trim + suppression du `;` final via `TrimSQL`) - loggue une erreur si le handle est nul (DB non ouverte) - en cas d’échec : appelle `HandleError` avec un extrait de la requête

```
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
```

Pour les batchs d'écritures :

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

QueryFast(selectSql)

`QueryFast` exécute un `SELECT` et renvoie un `Variant(2D)` prêt à coller :

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

À choisir par défaut : c'est direct, rapide, propre.

`SelectToSheet` passe par un CSV puis `QueryTables.Add`. C'est moins rapide que `QueryFast`, mais peut être utile si vous voulez exploiter le pipeline d'import Excel (encodage, parsing, etc.).

Recommandation : utilisez `QueryFast` pour les usages analytiques, `SelectToSheet` si vous avez une contrainte Excel spécifique.

Scalar : une valeur unique (COUNT/MAX/…) sans rapatrier un tableau

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

Le scalar est parfait pour :

- compteurs, bornes, flags
- contrôles avant export
- tests de présence

Démo : `Mod2DuckDb_Scalar.bas`

FrameFromValue

Crée une table "frame" DuckDB depuis un `Variant(2D)` (souvent `Range.Value2`) :

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

Cas d'usage : staging express pour faire immédiatement des joins/agg en SQL.

AppendArray(tableName, data2D, hasHeader)

Alimente une table DuckDB via **Appender** (très rapide), sans CSV/ADO :

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

Démo "dataframe-like" : `Mod2DuckDb_PythonLike.bas` (`Test_FrameFromValue_PythonLike`)

Fonctions Spéciales

UpsertFromArray(tableName, data2D, headerRow, keyColsCsv)

Utilisé **Excel comme UI de saisie/édition** pour mettre à jour/ insèrer data dans une base DuckDB selon une ou plusieurs clés sur feuille excel :

- **DB → Excel** : `ReloadFromDuckToExcel` (recharger une table en feuille)
- **Excel → DB** : `PushExcelToDuck` (pousser une zone Excel et faire un **UPSERT**)

Conditions:

1) La table DuckDB cible existe (`tableName`)

2) La feuille contient un bloc tabulaire à partir de `A1` :
   - **ligne 1 = en□têtes**
   - lignes suivantes = données
3) Les en□têtes Excel sont les **noms EXACTS** des colonnes DuckDB (même orthographe).
4) Les colonnes clés doivent être **présentes dans la zone Excel** et dans la table.
5) Les clés doivent être **non nulles** (pas de clé vide).
6) Idéalement, la clé (ou combinaison de clés) est **unique** dans Excel *et* en base.

Excel → DB : `PushExcelToDuck`

1) Ouvre `demo.duckdb` en lecture/écriture

2) Lit le bloc Excel depuis `A1` via :
   - `arr = ws.Range("A1").CurrentRegion.Value`
3) (Debug) affiche :
   - colonnes DB (`information_schema.columns`)
   - en□têtes Excel (ligne 1 du tableau)
4) Appelle :
   - `db.UpsertFromArray tableName, arr, 1, keyCols`
5) Relit la table et rafraîchit la feuille (contrôle visuel)

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

- `headerRow=1` : la ligne 1 contient les noms de colonnes.
- `keyColsCsv` : liste des clés (insensible à la casse, normalisée).

Cas d'usage : mise à jour de référentiels, historisation, synchronisation incrémentale.

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

Comment l'UPSERT fonctionne réellement (mécanique interne)

1) **Mapping colonnes Excel → colonnes DuckDB**
   - si `headerRow=1` : mapping par **noms** d'entêtes
2) **BEGIN TRANSACTION** (atomicité + perf)
3) **Création d'une table TEMP de staging**
   - ex : `temp.__tmp_upsert`
   - avec les colonnes utiles dans le bon ordre
4) **Ingestion ultra rapide** des lignes Excel dans la temp table (Appender)
5) UPSERT "SQL générique" (sans MERGE) :
   - **UPDATE … FROM temp** sur les clés
   - **INSERT … WHERE NOT EXISTS** pour les nouvelles lignes
6) Nettoyage : `DROP temp.__tmp_upsert`
7) **COMMIT** (ou ROLLBACK en cas d'erreur)

Gros filtres : Temp List pour remplacer WHERE IN (…)

Pour éviter un énorme `WHERE IN (...)`.

1) `CreateTempList "__basket", keys, "TEXT"` → crée `temp.__basket(v TEXT)` et l'alimente rapidement
2) SQL : `... WHERE isin IN (SELECT v FROM __basket)` **ou** `JOIN __basket ON ...`

Option A : "brique" (`CreateTempList` + SQL libre)

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

Option B : "clé-en-main" (`SelectWithTempList`)

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

Démos : - `Mod2DuckDb_WhereInSimple.bas` - `Mod2DuckDb_WhereInFct.bas`

Fonctions Dictionaire

Remplit un `Scripting.Dictionary` issue de données DuckDB, la requête doit retourner une colonne qui sert de clé.

- **DictFlat** : `clé → valeur` (1 colonne)
- **DictRow1D** : `clé → tableau 1D de valeurs` (plusieurs colonnes, *sans* labels)
- **DictRow2D** : `clé → tableau 2D [labels; valeurs]` (plusieurs colonnes, *avec* labels)

Quand plusieurs lignes partagent la même clé :

Paramètres :

- `onDupMode = 0` : **ignore** les doublons → conserve la **première occurrence**
- `onDupMode = 1` : **remplace** → conserve la **dernière occurrence rencontrée**
- `clearFirst=True` : `dict.RemoveAll` avant remplissage
- `clearFirst=False` : on "merge" (utile si tu construis le cache en plusieurs passes)

A) clé → valeur

- `SelectToDictFlat(selectSql, keyCol, [valCol]) As Dictionary`
- ou `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
```

Démo : `Mod2DuckDb_DictFlat.bas` (ex. `EX_ISIN_ToName_Dict`, `Test_DictFlat_Duplicates`)

B) Row1D : clé → tableau 1D des valeurs

`SelectToDictRow1D(selectSql, keyCol, dict, [valueColsCsv])`
Idéal si vous voulez plusieurs colonnes, accès par index.

Démo : `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 : clé → (titres + valeurs) en petit tableau 2×N

`SelectToDictRow2D(selectSql, keyCol, dict, ...)`
Plus lisible (titres inclus), un peu plus lourd, permet de rechercher la valeur associé à la clé selon le label de la table

Pour chaque clé : - `dict(key)` est un **tableau 2D** (bornes 1..2 × 1..(ncol-1)) : - ligne 1 : noms de colonnes (hors clé) - ligne 2 : valeurs correspondantes (typées)

Démo : `Mod2DuckDb_DictRow2D.bas`

Import de données

1) **Scan / Query direct** (zéro table persistante)

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

2) **CTAS** (*Create Table As Select*) : matérialise une table

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

3) **COPY FROM** : charge dans une table existante (souvent le plus “contrôlé”)

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

A) Lecture auto : ReadToArray(filePath, [tailSql])

- **CSV** : support natif (pas d’extension).
- **Parquet** : souvent disponible par défaut, mais votre toolkit prévoit aussi `db.TryLoadExt "parquet"` / `db.LoadExt "parquet"` (voir `Mod1DuckDb_Extension.bas`).
- **JSON** : support via l’extension `json` (souvent auto□chargée, mais vous pouvez forcer : `db.TryLoadExt "json"` / `db.LoadExt "json"`).

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

Le wrapper choisit la bonne fonction selon l’extension (Parquet / JSON / CSV auto).

B) CSV

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

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

`read_csv_auto` Auto□détecte delimiter/quote/header/types, avec des options de robustesse.

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

Options fréquentes

```
SELECT *
FROM read_csv_auto(
  'data.csv',
  delim=';',
  header=true,
  sample_size=-1,          -- lire + pour inférer le schéma
  ignore_errors=true,      -- skipper les lignes "cassées"
  nullstr=['', 'NULL']     -- valeurs à interpréter comme NULL
);
```

`read_csv` (schéma explicite / contrôle fort) Utile quand vous **connaissez** les types ou quand l’auto□détection est instable.

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

import propre dans une table

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

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

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

C) Parquet (recommandé pour gros volumes)

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

Démos : - `Mod1DuckDb_Parquet.bas` - `Mod1DuckDb_Parquet_Research.bas`

`read_parquet` **(standard)**

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

Replacement scan (DuckDB infère via l'extension)

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

Import table (CTAS)

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

Lire plusieurs fichiers / datasets

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

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

D) JSON : NDJSON ou JSON array

- `cDuck.CopyToJson(selectSql, outJson, overwrite:=True, boolJsonArray:=False)`
  - `boolJsonArray=False` → **NDJSON** (1 objet par ligne)
  - `boolJsonArray=True` → **JSON Array** (`[ {...},{...} ]`)
- `cDuck.CopyToJsonx(selectSql, outJson, overwrite:=True)`
  - force `FORMAT JSON` (NDJSON par défaut).

Démos : `Mod1DuckDb_Json.bas`

`read_json_auto` **(votre 1er réflexe)**

DuckDB sait souvent deviner tout seul (records / format / nesting).

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

Contrôler le format** (ndjson vs array vs unstructured)**

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

Contrôler le “unpack” des objets (`records`)

- `records=true` : transforme les clés JSON en colonnes.
- `records=false` : conserve le JSON dans une colonne `STRUCT/JSON`.

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

NDJSON (default)

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

JSON Array (souvent demandé par des APIs)

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

Export d’un SELECT

```sql
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 méthodes)

1) **ODBC** `odbc_query` : Access exécute la requête, DuckDB récupère le résultat
2) **ODBC** `odbc_scan` : copie “table brute” depuis Access vers DuckDB
3) **ADO Recordset** → `AppendAdoRecordsetFast` : ingestion rapide via la DLL bridge
4) **ADO Recordset** → **Variant(2D)** → `AppendArray` : fallback universel, plus RAM

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

Méthode	Qui exécute le “gros du travail” ?	SQL à écrire	Points forts	Limites / risques	Pré-requis
ODBC / `odbc_query`	**Access/ACE** interprète la requête ; DuckDB récupère le résultat	SQL **Access/ACE** (avec `[]`, fonctions Access, JOIN…)	• Très flexible • Souvent **très rapide** si filtres/jointures côté Access	• Plus “fragile” (dialecte SQL Access)• Erreurs driver possibles• Pas du SQL DuckDB	• Extension DuckDB `odbc` / `nanodbc` • + **driver ODBC Access**
ODBC / `odbc_scan`	**DuckDB** (via extension ODBC)	SQL **DuckDB** autour (`WHERE`, `LIMIT`…)	• Simple, robuste pour **copie table brute**	• Peu flexible côté Access• Filtres/jointures plutôt côté DuckDB	• Extension DuckDB `odbc` / `nanodbc` • + **driver ODBC Access**
ADO Recordset → `AppendAdoRecordset` *(DLL bridge)*	**DLL bridge** lit le Recordset + insère dans DuckDB	SQL Access (dans `rs.Open`)+ DuckDB (table cible)	• Pas besoin extension ODBC DuckDB• Bonnes perfs• Pas de gros array VBA	• Dépend d’ACE OLEDB• Réglages curseur/forward-only importants	• **Microsoft.ACE.OLEDB.12.0**• + `duckdb.dll` • + `duckdb_vba_bridge.dll`
ADO Recordset → Variant(2D) → `AppendArray` *(DLL bridge)*	**VBA** matérialise le Variant(2D), puis **DLL bridge** ingère via `AppendArray`	SQL Access (dans `rs.Open`)+ DuckDB (table cible)	• Fallback pratique• Tu peux inspecter/afficher l’array• Ingestion finale reste en **DLL**	• Plus de **RAM** (tout en mémoire VBA)• Souvent plus lent sur gros volumes	• ACE OLEDB• + `RecordsetToVariant2D` • + `duckdb_vba_bridge.dll` (pour `AppendArray`)

Export de données

Export CSV

A) Export d’une table

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

B) Export d’un SELECT

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

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

C) Options utiles (locale / Excel)

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

Export Parquet

Export simple

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

Options de perf / format

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

Écriture partitionnée (dossier lakehouse)

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

Métadonnées & maintenance (catalogue, renommages, checks)

Fonctions principales (cDuck) :

- `TablesInfo([schema])` : liste tables/vues
- `ColumnsInfo(table)` : colonnes/types
- `TableExists("schema.table")` / `ColumnExists(...)`
- `RenameTable`, `RenameColumn`
- `SelectShape` : dimensions d'un SELECT sans rapatrier les données

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

Extensions DuckDB

DuckDB a été conçu comme un moteur **embarquable** (*in□process*) et **extensible** : une partie importante des fonctionnalités n'est pas "hardcodée" dans le noyau, mais livrée sous forme **d'extensions** chargeables à la demande. Une extension est un **binaire** que DuckDB charge dynamiquement pour activer de nouveaux formats, fonctions, scanners, etc.

- Vue d'ensemble (docs) : https://duckdb.org/docs/stable/extensions/overview
- Portail extensions communautaires : https://duckdb.org/community_extensions/

Deux familles : *Core extensions* vs *Community extensions*

- **Core extensions** : extensions "officielles", distribuées et testées avec les releases DuckDB (niveau de support variable selon l'extension).
- **Community extensions** : extensions développées/maintenues par la communauté (qualité variable, mais souvent excellentes).

DuckDB met en cache les binaires d'extension dans un répertoire utilisateur, typiquement sous :

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

un upgrade DuckDB (`v1.4.3` → `v1.4.4`) change le sous-dossier de version.

Côté DuckDB VBA :

- `INSTALL <ext>;` télécharge/installe l'extension dans le cache local.
- `LOAD <ext>;` charge l'extension dans la session courante (donc active les fonctions).
- soit exécuter ces commandes (`db.Exec "INSTALL …"; db.Exec "LOAD …"`)
- soit utiliser ton wrapper `db.LoadExt "<ext>"` (recommandé), qui encapsule le best-effort.

Quand DuckDB est mis à jour, il arrive que l'extension déjà en cache ne corresponde plus (ABI / signature / version). La parade la plus simple :

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

Extensions Core (officielles)

- Maintenues par l'équipe DuckDB.
- Distribuées via le dépôt officiel.
- Exemples typiques : `parquet` , `json` , `httpfs` , `ui` (selon la distribution / la doc).

Extensions "Community"

- Contribuées par des tiers (non maintenues par DuckDB Labs).
- Installables via le dépôt "community extensions".
- Exemples dans tes modules : `miniplot` , `rapidfuzz` , `nanodbc`

miniplot `miniplot` ajoute des fonctions SQL qui génèrent des **graphiques** (souvent en HTML/JS) directement depuis DuckDB.

Côté Excel/VBA : - tu produis un graphique en SQL - tu récupères soit un **fichier HTML généré** (si supporté par la fonction) - soit un **fragment HTML renvoyé** en résultat SQL, que tu écris toi-même dans un fichier.

- `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 apporte des fonctions SQL performantes (lib C++) pour :

- distance d'édition
- similarité
- variantes (Jaro-Winkler, prefix/postfix, partial match…)

Puis scorer uniquement les candidats restants :

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

- `rapidfuzz_ratio(a,b)` : similarité "générale" (score)
- `rapidfuzz_partial_ratio(a,b)` : sous-chaînes (utile pour "contains" tolérant)
- `rapidfuzz_jaro_winkler_*` : très bon pour noms propres / fautes de frappe courtes
- `rapidfuzz_prefix_*` / `rapidfuzz_postfix_*` : pour comparer préfixe/suffixe
- `rapidfuzz_osa_*` : distance OSA (gère transpositions adjacentes)

nanoODBC DuckDB charge une extension ODBC qui lui permet de :

- se connecter à n'importe quelle source ayant un driver ODBC
- lire des tables / exécuter des requêtes
- matérialiser les résultats dans DuckDB

Deux modes : `odbc_scan` vs `odbc_query`

odbc_scan (copie brute)

- Tu donnes un `table_name`
- DuckDB lit "tel quel"
- Super pour du staging / migration table entière

odbc_query (requête poussée côté source)

- Tu fournis un SQL **interprété par Access/ACE** via ODBC
- Ultra intéressant si tu veux filtrer/joindre côté Access avant transfert

Heuristique simple :

- **scan** = "copie brute"
- **query** = "extraction intelligente" (pushdown)
- `LoadExt(name)` : charge (INSTALL + LOAD côté DLL)
- `TryLoadExt(name) As Boolean`
- `EnsureOdbcLoaded()` : tente `odbc`, sinon `nanodbc`

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

Extension UI : interface web locale

L'extension `ui` démarre un petit serveur HTTP local et ouvre une UI dans le navigateur.

Idéal pour : - inspecter le catalogue, les tables, les requêtes - prototyper du SQL sur une base `.duckdb` - debug "data engineering" sans quitter Excel

Tu as fait le bon choix technique :

- `gUiDuck` global pour garder la connexion vivante
- ouverture en **lecture seule** (`OpenReadOnly`) pour sécuriser
- `CALL start_ui();` pour démarrer
- `CALL stop_ui_server();` + close pour arrêter

5. Explication du code

Notes techniques du DLL

- **Interface :** Unicode (UTF-16) côté VBA, conversion UTF-8 coté DLL
- **Chaînes** : toutes les fonctions …W consomment des String UTF-16 passées via StrPtr(…).
- **Erreurs** : retour 0 = échec. Détail lisible via Duck_LastErrorW (buffer UTF-16).
  - La plupart des fonctions renvoient `int` : **1 = succès**, **0 = échec*, en cas d'échec, un message est stocké dans un buffer d'erreur thread-local
- **Normalisation d'identifiants** : le SQL est pré-traité — les identifiants […] et `...` sont convertis en "…", les littéraux '…' et "…" restent intacts.
- **Chemins** : les backslashes vers / sont normalisés côté DLL (utile dans COPY TO).
- **Nettoyage** : Toute ressource DuckDB (`result` , `prepare` , `appender`) et COM (`SAFEARRAY` , `VariantClear`) est systématiquement libérée.
- **Dates/Heures**
  - OLE Automation date (VT_DATE) via `SystemTimeToVariantTime` / `VariantTimeToSystemTime` .
  - Time-only mappé en VT_DATE relatif au 1899-12-30 (convention Excel).
- Appenders DuckDB utilisés pour ingestion rapide (plus rapide que `INSERT`).
- Transactions `BEGIN/COMMIT` automatiques dans les fonctions UPSERT et Appender.
- **Unicode / Wide strings** : `...W` et `StrPtr` , toutes les fonctions suffixées `W` attendent des chaînes **UTF-16** (Wide),c côté VBA, on passe **un pointeur** sur la chaîne : `StrPtr(s)` → `LongPtr`

SAFEARRAY : - SelectFill2D_TypedV attend un Variant(2D de Variant) déjà dimensionné (ligne 1 = entêtes). - QueryToArrayV alloue et remplit un Variant(2D) (ligne 1 = entêtes). - Les tableaux sont column-major

- **Typage (retours SELECT)** :
  - Nombres entiers → **VT_R8** (compat Excel), flottants → **VT_R8**.
  - `DATE/TIME/TIMESTAMP` → **VT_DATE** (base OLE).
  - `VARCHAR/TEXT/DECIMAL/UUID/INTERVAL` → **VT_BSTR**.
  - `BLOB` → **SAFEARRAY(Byte)** 1D (VT_ARRAY|VT_UI1).
- **Appender** : ingestion **RAM** ultra-rapide (via `AppendArrayV`) sans CSV ni ADO.
- **Extensions** : `DuckVba_LoadExtW` fait **INSTALL** + **LOAD** (les erreurs d'INSTALL « déjà installé » sont ignorées).

Ouverture / fermeture & exécution 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
```

- Objectif : Ouvre une base DuckDB en lecture/écriture ou lecture seule (fichier .duckdb ou ":memory:") et retourne un handle de connexion.
- Entrée : dbPath = pointeur vers la chaîne du chemin.
- Retour : LongPtr non nul = handle ; 0 si échec (lire Duck_LastErrorW).
- Usage : `h = DuckVba_OpenW(StrPtr("C:...\cache.duckdb"))`

Arguments - `pwszPath` : pointeur vers `UTF-16` (Wide).

Exemples : - `":memory:"` → base en RAM - `"C:\...\my.duckdb"` → base fichier - chemins UNC possibles (`\\server\share\...`) mais attention locks/latences.

Vigilences - DB fichier : peut être verrouillée par un autre process. - Ne pas oublier `DuckVba_Close`. - Ne pas partager un handle entre threads (VBA n'est pas thread-safe).

DuckVba_OpenReadOnlyW

Ouvre une base en lecture seule.

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

Arguments - `pwszPath` : chemin DB wide.

DuckVba_Close

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

- Objectif : Ferme proprement la connexion associée au handle et libère les ressources.
- Retour : 1 si OK, 0 sinon.

Arguments - `h` : handle de session.

DuckVba_ExecW

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

- Objectif : Exécute du SQL arbitraire (DDL/DML, COPY, etc.).
- Retour : 1 si OK, 0 sinon (détail via Duck_LastErrorW).
- Remarques : Le pont réécrit automatiquement les identifiants `[...]` et `...` en guillemets doubles "…" pour coller à DuckDB.

Arguments - `h` : session ouverte - `pwszSql` : SQL wide. - accepter multi-statements ? (*inféré*) souvent oui, mais mieux de rester 1 statement/Exec. - `;` final généralement optionnel.

Cas d'usage - DDL : `CREATE TABLE`, `DROP`, `ALTER` - DML : `INSERT`, `UPDATE`, `DELETE` - control : `BEGIN`, `COMMIT`, `ROLLBACK` - export : `COPY (...) TO ...`

DuckVba_QueryToArrayFastV

Exécute un SELECT et remplit un `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` : SELECT wide - `v` : out param ; reçoit un tableau 2D.

Duck_LastErrorW

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

- Objectif : Récupérer le dernier message d'erreur (UTF-8 natif → UTF-16 dansbuffer).
- Retour : Nb de caractères copiés (hors NUL). 0 si rien.

Arguments - `pwszBuf` : buffer wide *pré-alloué* côté VBA. - `cch` : capacité en caractères.

DuckVba_FrameFromValue

Crée une table "frame" DuckDB depuis un `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` : nom de la table (temp ou persistante selon `makeTemp`).
- `v` : tableau 2D.
- `hasHeader` :
  - `1` : première ligne = noms de colonnes.
  - `0` : colonnes auto-nommées (probable : col1, col2…).
- `makeTemp` :
  - `1` : table temporaire (scope session)
  - `0` : table "normale" (persistante dans la DB) — selon mode d'ouverture

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
```

- Objectif : Alimente une table DuckDB directement depuis un Variant(2D) (p.ex. Range.Value2) sans CSV.
  - Paramètres :
    - V : tableau 2D ; si hasHeader=1, la 1ère ligne est ignorée (considérée comme entêtes).
    - Mapping types : BOOL → bool, nombres → DOUBLE/INT, DATE → TIMESTAMP, texte → VARCHAR, Byte() → BLOB. Quand : Push massif en RAM très rapide grâce à l'Appender.

Cas d'usage - charger une extraction Excel vers DuckDB pour faire des joins/agg - staging table `__tmp__` pour upsert - etc

DuckVba_ScalarV

Exécute un SELECT qui renvoie une seule valeur (1 row, 1 col) et retourne ce 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` : typiquement `SELECT COUNT(*) FROM ...`

Retour

- `<>0` OK, `0` KO
- `v` : valeur (Variant), peut être `Null/Empty` si résultat vide.

DuckVba_LoadExtW

Charge une extension DuckDB.

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

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

Cas d'usage

- activer `read_parquet`, `parquet_schema`, `read_json_auto`
- activer `odbc_scan`, `odbc_query` pour Access/ODBC

Vigilence

- fichiers extension doivent être au bon endroit (répertoire attendu par DuckDB).

Info

DuckVba_TableInfoV

Retourne des infos "catalogue" sur tables/vues.

```
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 → tous schémas
  - sinon `"main"`, `"temp"`, etc.

DuckVba_ColumnsInfoV

Retourne metadata des colonnes d'une 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

Teste l'existence d'une table/vue.

```
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"

Retour

- `<>0` existe, `0` n'existe pas (ou erreur).

DuckVba_ColumnExistsW

Teste l'existence d'une colonne dans une 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` : colonne

Retour

- `<>0` existe, `0` n'existe pas (ou erreur)

DuckVba_RenameTableW

Renomme une 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` : peut inclure le schéma
- `pwszNewTable` : idem

Retour

- `<>0` OK, `0` KO

DuckVba_RenameColumnW

Renomme une colonne.

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` : colonnes

Retour

- `<>0` OK, `0` KO

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
```

- Objectif : Exécute un SELECT et écrit directement un CSV (UTF-8, entête).

- Retour : 1 si OK.
- Quand : Pour charger ensuite via QueryTables sans repasser par ADO.

Arguments

- `pwszSelect` : SELECT wide
- `pwszCsv` : chemin CSV wide

DuckVba_SelectShapeW

Retourne dimensions du résultat (sans renvoyer les données).

```
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)

Variante “typed” de SELECT→Variant, visant à préserver mieux les types (Date/Number) au lieu de tout en texte.

Cas d’usage

- quand Excel doit reconnaître dates / nombres sans parsing.
- export intermédiaire où le typage compte (dates, timestamps).

DuckVba_ReadCsvToTableW

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

- Si la table n’existe pas :
  - create_if_missing = 1 → CREATE TABLE AS SELECT * FROM read_csv_auto(…)
  - create_if_missing = 0 → erreur “table not found”
- Si la table existe : append via COPY table FROM ‘file’ (AUTO_DETECT=TRUE, HEADER=TRUE)

DuckVba_ExecPreparedToArrayV

Exécute un prepared statement et renvoie un `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` : handle prepared (retourné par `DuckVba_PrepareW`)
- `v` : out result

Retour

- `<>0` OK, `0` KO

DuckVba_PrepareW

Prépare un statement paramétré (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` : SQL wide, typiquement :
  - `"INSERT INTO t VALUES (?, ?, ?);"`
  - `"SELECT * FROM t WHERE id=?;"`

Retour

- `ps` handle prepared, ou `0` si KO.

Cas d'usage

- éviter injection : ne concatène pas des valeurs utilisateur
- stabiliser le plan : meilleures perfs dans les boucles

DuckVba_ExecPrepared

Exécute le prepared statement (sans retour tableau).

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

Cas d'usage

- INSERT/UPDATE paramétrés
- DDL paramétrable (moins courant)

DuckVba_Finalize

Libère le prepared statement.

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

DuckVba_CopyToParquetW

Exporte un SELECT en 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` : SELECT wide
- `pOut` : path parquet wide

Retour

- `<>0` OK, `0` KO.

Cas d'usage

- produire un dataset "lakehouse" depuis Excel/Access
- archiver des snapshots compressés
- partager des données volumineuses sans Excel

Fonctions Spéciales

DuckVba_AppendAdoRecordset

Objectif : Déduit le schéma à partir des Fields (ADO) et lit une table(MS Access VIA COM) → CREATE TABLE optionnel, et **ingérer** dans DuckDB par 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` : table cible
- `createIfMissing` :
  - `1` : crée table si absente (types dérivés du schema ADO)
  - `0` : table doit exister

Techniques COM

- Résout `Fields`, `Fields.Count`, `Fields.Item(i)`, puis sur `Field` : `Name`, `Type`, `Value`.
- **Infère le schéma** DuckDB (`ado_type_to_duck`) :
  `adBoolean→BOOLEAN, adBigInt→BIGINT, Single/Double→DOUBLE, Currency→DECIMAL(19,4), Date/DateTime→DATE/`
- `create_if_missing=1` → `DROP TABLE IF EXISTS` + `CREATE TABLE "name"(…)` (quoting sûr).
- Boucle lignes : `MoveFirst` best-effort puis `EOF` / `MoveNext`. Mapping ADO→duckdb_appender_* analogue à `AppendArrayV`.

DuckVba_AppendAdoRecordsetFast

Variante optimisée pour gros volumes.

Technique

- bulk fetch (GetRows) ou streaming forward-only, moins d'aller-retours COM.

DuckVba_UpsertFromArrayV

Upsert (merge) depuis un `Variant(2D)` vers une table cible, selon une ou plusieurs clés.

```
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
```

Objectif :

- Met à jour ou insère les lignes d'un tableau Excel(Mapping colonnes par noms d'entêtes (ligne header_row) sinon position) dans une table DuckDB,
- UPSERT **sans MERGE** (compatible versions □ SQL générique).
- Effectue un `UPDATE … FROM` puis un `INSERT … WHERE NOT EXISTS` et Transaction `BEGIN … COMMIT + DROP` de la table temp.
- Les colonnes clé doivent être présentes dans les colonnes mappées (sinon la jointure ne peut pas s'effectuer).

Arguments

- `pwszTable` : table cible
- `v` : tableau 2D contenant au moins les colonnes requises (incluant clés)

- `headerRow` : index de la ligne d'entête dans `v`
  - souvent `1` si la ligne 1 contient les noms de colonnes
- `pwszKeyColsCsv` : liste CSV de colonnes clés (wide), ex :
  - `"ISIN"`
  - `"ISIN,NumeroContrat"`

Pipeline précis 1) **Découverte des colonnes cibles**
- `SELECT column_name FROM information_schema.columns WHERE schema/table` .
- Construction `tgt_cols[]` + `tgt_cols_norm[]` (via `normalize_ident_w`) pour comparaisons case-insensibles.

2) **Parsing des clés**
   - `key_columns_csv_w` → segments (`,`), `trim_ws_dup` + `normalize_ident_w` .
   - Vérification présence dans la sélection.

3) **Mapping colonnes Excel → table**
   - Si `header_row>=1` ET dans les bornes → lecture entêtes (BSTR) et normalisation, **match par nom**.
   - Complément **positionnel** sans collision (ex: si certaines colonnes non nommées).
   - Construction `sel_tgt_idx[]` et `sel_xls_idx[]` (seulement colonnes utilisées).

4) **BEGIN TRANSACTION** (atomicité + perfs).

5) **CREATE TEMP TABLE __tmp_upsert AS SELECT FROM WHERE 0;**
   - `<qtab>` = `"schema"."table"` (quoting sûr).
   - Temp table **structure identique** à la sélection.

6) **Remplissage TEMP via Appender**
   - Parcours des lignes utiles (`rstart = header_row+1` si entêtes).
   - Mapping **COM→DuckDB** identique à `AppendArrayV` .

7) **UPDATE…FROM + INSERT manquant**
   - `is_key[i]` pour identifier colonnes de jointure.
   - **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>`
   - Construction des fragments : `set_list` , `where_update` , `where_exists` , `ins_cols` , `ins_vals` (quoting identique).

8) **DROP TEMP + COMMIT**
   - `DROP TABLE IF EXISTS temp.__tmp_upsert` (best-effort).
   - COMMIT, sinon ROLLBACK en cas d'erreur.

DuckVba_CreateTempListV

Objectif : Crée une table TEMP tabname(v) et la remplit à partir d'un Variant (1D ou 2D → 1ère colonne).

```
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 orientatif (VARCHAR, INT, DOUBLE…) ; la DLL convertit les valeurs.
- Quand : Pour faire WHERE x IN (SELECT v FROM tabname) ou JOIN tabname t ON ….
- Atout : Très efficace pour des listes de plusieurs milliers d'IDs par rapport à un IN (…) géant.

Technique

- Détecte le "genre" cible (`TMP_VARCHAR` / `TMP_INT64` / `TMP_DOUBLE`) via `sql_type` (match simple sur mots-clés).
- **Appender** préféré ; fallback **prepared INSERT** si l'appender échoue (schema "temp" indispo → réessaie "main").

Arguments

- `tabname_w` : nom de table temp (wide)
- `keys` :
  - attendu : liste de valeurs (Variant 1D, ou 2D 1-col)
  - types : Strings/LongLong/Double selon `sqltype_w`
- `sqltype_w` : type SQL explicite (ex `"VARCHAR"` , `"BIGINT"` , `"DOUBLE"`)
  - sert à forcer le type des clés côté DuckDB, éviter inference bancale

DuckVba_SelectWithTempList2V

API "clé-en-main" : crée/alimente une temp list et renvoie un SELECT filtré/jointé.

```
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` : nom table de clés
- `keys` : valeurs clés (Variant)
- `pSqlType` : type DuckDB des clés (wide)
- `pSelectOrTable` :
  - soit un **nom de table** (ex `"main.clients"`)
  - soit une **requête SELECT** (ex `"SELECT * FROM ..."`).
- `pJoinCol` :
  - pointeur wide vers la colonne de join, ou `0` /NULL si non fourni.
- `autoJoin` :
  - `1` : le DLL essaie d'auto-déterminer la jointure (*inféré* : par nom de colonne commun, ou première colonne, ou colonne unique)
  - `0` : jointure explicite requise via `pJoinCol` .
- `vOut` : résultat en Variant(2D)

Technique interne

1) crée temp list (comme `CreateTempListV`)
2) construit un SQL du style :
   - `SELECT ... FROM (<selectOrTable>) t JOIN <temp> k ON t.<joinCol>=k.key`

3) exécute et convertit en Variant(2D)

DuckVba_SelectToDictW

Remplit un Dictionary où chaque clé pointe vers une **ligne** (souvent un tableau 1D/2D) (selon impl).

```
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
```

- Objectif : Exécute un SELECT et remplit un Scripting.Dictionary fourni (via ObjPtr(dict)).
- Clé : la colonne keyCol (nom insensible à la casse, tolère [], ‘ et “).
- Valeur : un SAFEARRAY 2×J :
- Ligne 1 = titres (toutes colonnes sauf la clé)
- Ligne 2 = valeurs (typées : BOOL/DOUBLE/DATE/BSTR/BLOB)
  - Options :
    - clearFirst=1 → dict.RemoveAll avant.
      - onDupMode=0 → ignore doublons ; 1 → remplace (Item(key) = val).
- Quand : Idéal pour un accès clé→ligne en VBA (ex. D(“FR0002”)).

Arguments (détails)

- `pwszSelect` : SELECT qui retourne au moins la colonne clé
- `pwszKeyCol` : nom de la colonne clé (wide)
- `pDict` : `ObjPtr(dict)` où `dict` est un `Scripting.Dictionary`
- `clearFirst` :
  - `1` : `dict.RemoveAll` avant
  - `0` : append/merge
- `onDupMode` :
  - **probable** : `0` ignore les doublons, `1` remplace l’existant

DuckVba_SelectToDictFlatW

Remplit un Dictionary clé → valeur** (flat).**

```
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` :
  - peut être un pointeur NULL (0) pour “auto” si le SELECT retourne exactement 2 colonnes (**observé dans ta classe** via `pVal=0&`).
  - sinon : nom de colonne valeur.

DuckVba_SelectToDictValsColsW

Remplit un Dictionary clé → “valeurs de certaines colonnes”.

```
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 → “toutes colonnes sauf la clé” (comportement suggéré par ton wrapper) - sinon `"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);
```

Annexe

Fichier Parquet

> **Un fichier Parquet est un format de fichier colonnaire (column-oriented) optimisé pour stocker et traiter de gros volumes de données.**

Parquet est un format de stockage **colonnaire** et **binaire** introduit en 2013 (initié notamment par Cloudera et Twitter), conçu pour l'analytique à grande échelle, en particulier dans les environnements Hadoop/Spark et les architectures **data lake / lakehouse**. Contrairement au CSV, qui organise les données par **lignes**, Parquet les stocke par **colonnes**. Cette organisation colonnaire permet aux moteurs de requêtes de lire uniquement les colonnes nécessaires (au lieu de balayer toute la table), tout en profitant d'une compression et d'un encodage beaucoup plus efficaces.

Avec Parquet, il n'est généralement plus nécessaire de charger l'intégralité d'une table en mémoire pour l'analyser : le moteur lit uniquement les blocs et colonnes utiles à la requête, en exploitant les métadonnées et le schéma stockés dans le fichier. Les données sont lues uniquement là où elles sont utiles.

Limite CSV

Les formats de fichiers "plats" classiques comme CSV privilégient la simplicité et la portabilité, mais présentent des limites importantes en analytique :

- Stockage en texte (parsing coûteux, ambiguïtés de types),
- Organisation orientée lignes, qui implique de lire une grande quantité de données inutiles lorsque seules quelques colonnes sont requises.
- Absence de compression : Les fichiers plus volumineux peuvent entraîner une augmentation des coûts de stockage et de transfer.
- Faible capacité d'optimisation côté moteur (peu ou pas de métadonnées exploitables).

Parquet répond à ces limites en proposant un format binaire et colonnaire, pensé pour les moteurs de requêtes distribués.

Format colonne (Parquet)

Parquet stocke les valeurs colonne par colonne, **regroupées en blocs**(row groups / column chunks) et accompagnées de métadonnées (schéma, statistiques, encodage, compression). Résultat : les moteurs peuvent limiter la lecture aux colonnes réellement utilisées (**column pruning**) et parfois éviter de lire certains blocs (predicate pushdown), ce qui accélère fortement les traitements analytiques.

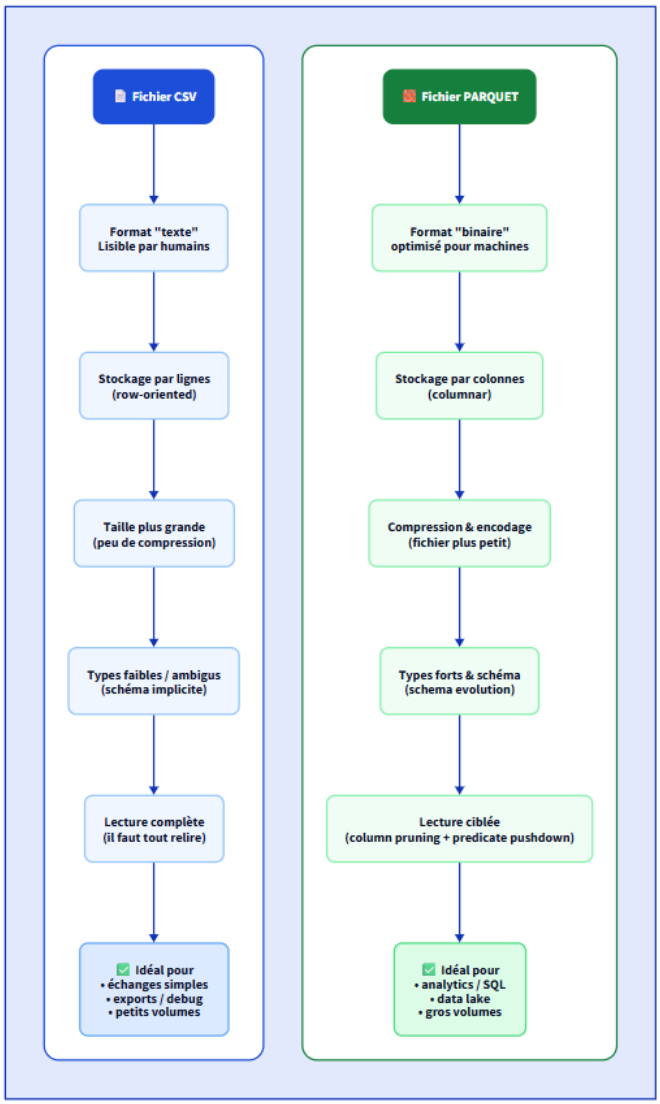
- **Gain d'espace** : Un dataset de 1 To en CSV réduit à 130 Go en Parquet, soit 87% de gain
- **Gain de vitesse** : une requête SQL passant de 236 s (CSV) à 6,78 s (Parquet), soit environ ×34 plus rapide, selon le moteur et le type de requête.

Structure interne (principes techniques)

Les fichiers Parquet sont divisés en groupes de lignes, qui contiennent un lot de lignes. Chaque groupe de lignes est divisé en morceaux de colonnes, chacun contenant les données d'une colonne. Ces blocs sont ensuite divisés en morceaux plus petits appelés pages, qui sont comprimés pour économiser de l'espace. De plus les fichiers Parquet contiennent des informations supplémentaires dans le pied de page, appelées métadonnées, qui permettent de localiser et de lire uniquement les données dont nous avons besoin.

- **Typage des données (schema)** : Parquet est un format binaire qui embarque le schéma (types : int, float, bool, date…). Contrairement au CSV (tout en texte), cela réduit les conversions, les ambiguïtés (dates, séparateurs) et les erreurs de typage.

- **Compression** : Parquet supporte plusieurs codecs (Snappy, Gzip, Zstd…). Comme les valeurs d'une même colonne se ressemblent souvent, la compression est généralement très efficace, ce qui diminue fortement l'espace disque et le volume de données à lire.
- **Encodage (data encoding)** : Avant ou en plus de la compression, Parquet applique des encodages colonnaires (ex. Dictionary Encoding pour les valeurs répétitives, RLE, bit-packing). Résultat : données plus compactes et décodage plus rapide.
- **Lecture/écriture parallèle** : Les fichiers Parquet sont découpés en blocs internes (row groups / pages), ce qui permet aux moteurs distribués (ex. Spark) de lire et traiter les données en parallèle, accélérant les traitements sur gros volumes.
- **Métadonnées enrichies (optimisation des requêtes)** : Parquet stocke des métadonnées et des statistiques par bloc (ex. min/max, nombre de valeurs nulles). Les moteurs SQL peuvent s'en servir pour éviter de lire des blocs non pertinents lors d'un filtre (predicate pushdown), réduisant le scan et améliorant la performance.



OLTP vs OLAP

OLTP

OLTP signifie **Online Transaction Processing** (traitement transactionnel en ligne). C'est une base conçue pour faire fonctionner une application au quotidien. Elle doit traiter beaucoup de petites opérations rapides (lire/modifier une fiche, créer une commande, valider un paiement) avec une forte garantie d'intégrité (transactions ACID) et une **latence minimale**. Elle est donc optimisée pour accéder à un petit nombre de lignes via des index, et s'appuie le plus souvent sur un stockage **orienté lignes** (row-store), adapté aux lectures/écritures ciblées.

Détails techniques

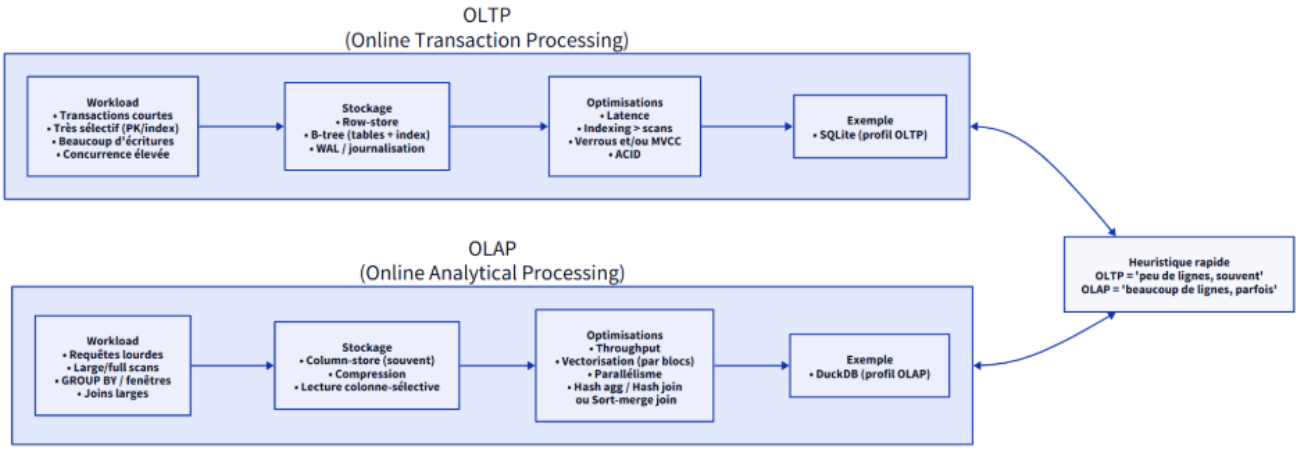
Le modèle de stockage est généralement row-store avec structures **B-tree** (données + index), car on lit/écrit souvent plusieurs colonnes d'une même ligne. Les mécanismes clés sont l'**ACID**, le contrôle de concurrence (verrous, MVCC selon le SGBD), le **write-ahead logging (WAL)**, et des plans de requêtes qui privilégient l'indexing plutôt que le scan. Les schémas sont souvent normalisés pour limiter la redondance et maintenir la cohérence lors des mises à jour.

OLAP

OLAP signifie **Online Analytical Processing** (traitement analytique en ligne). C'est une base conçue pour **analyser et agréger des données**. Elle exécute des requêtes moins fréquentes mais bien plus lourdes (scans, GROUP BY, fenêtres, joins larges) sur de **grands volumes** afin de produire des indicateurs, des rapports ou de l'exploration ad hoc. Elle est optimisée pour lire beaucoup de lignes en ne touchant que certaines colonnes, ce qui favorise un stockage **orienté colonnes** (column-store), avec **compression**, exécution **vectorisée** et souvent **parallélisme** pour maximiser le débit.

Détails techniques

C'est un moteur optimisé pour des requêtes **CPU/mémoire intensives** sur de **grands volumes**, avec un objectif principal de débit (throughput) plutôt que la latence unitaire. Le workload typique implique des full/large scans, des agrégations, des joins larges, du fenêtrage et de la lecture sélective en colonnes. D'où l'usage fréquent d'un stockage **column-store**, qui permet **compression**, meilleure **localité cache**, et exécution vectorisée (traitement par blocs) ; les agrégations profitent aussi d'algorithmes comme le hash aggregation et les joins de type **hash join/sort-merge join** selon les distributions. Les données sont souvent organisées en schémas **dénormalisés** ou en **étoile** (facts/dimensions) pour réduire le coût des joins et accélérer les agrégations.



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

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

SQLite vs DuckDB

Critère	Transactionnelle (OLTP)	Analytique (OLAP)
Objectif	Traitement de transactions unitaires en temps réel	Traitement analytique massif sur données historiques
Modèle de stockage	Basé sur les lignes	Basé sur colonnes
Exécution de la requête	À base d'itérateurs (ligne par ligne)	Vectorisé (par lots)
Modèle de données	Fortement normalisé (3NF, contraintes d'intégrité fortes)	Dénormalisé (schéma en étoile ou flocon, vues matérialisées)
Type d'accès	Accès aléatoire (lecture/écriture de quelques lignes)	Accès séquentiel (lecture de millions de lignes, peu d'écritures)
Type de requêtes	CRUD rapides (`INSERT` , `UPDATE` , `DELETE` , `SELECT` simple)	Agrégations lourdes, jointures multiples, fonctions analytiques
Structure du stockage	Row-store (stockage par lignes)	Column-store (stockage par colonnes, souvent compressé)
Mécanisme d'indexation	Index B-Tree, hash, full-text, pour accélérer les requêtes unitaires	Index bitmap, dictionnaires, compression vectorisée
Gestion de transactions	ACID strict, faible latence, isolation élevée	Moins de transactions simultanées, plus de lecture en batch
Concurrence	Haute (verrouillage fin, MVCC, nombreux threads courts)	Moyenne (parallélisation CPU, traitement vectorisé)
Écriture	Fréquente, petites opérations atomiques	Rare, chargement par lots (batch ETL, append-only)
Lecture	Sélective, par clé primaire ou index	Massivement séquentielle, scans complets optimisés
Performance cible	Latence minimale (ms)	Débit maximal (Mo/s → Go/s)
Caching	Cache de pages / buffer pool	Cache de blocs en colonnes, compression, vectorisation CPU
Taille typique de dataset	Mégaoctets à quelques gigaoctets	Gigaoctets à téraoctets
Fréquence des opérations	Très élevée (milliers d'opérations/s)	Faible (requêtes lourdes mais peu fréquentes)
Historisation	Données "vivantes" et courtes (transactions courantes)	Données historiques, souvent en lecture seule
Optimisation du plan d'exécution	Optimisé pour l'accès indexé et les transactions rapides	Optimisé pour les scans de colonnes et les agrégations parallèles
Technologies typiques	MySQL, PostgreSQL, SQLite, SQL Server, Oracle	DuckDB, ClickHouse, Snowflake, BigQuery, Redshift, Druid
Cas d'usage	ERP, e-commerce, systèmes de réservation, applications mobiles	Data warehouse, BI, reporting, analyse exploratoire
Type d'utilisateur	Application temps réel, utilisateurs finaux	Analystes, data engineers, data scientists
Architecture typique	Bases réparties OLTP, réplication maître/esclave	Entrepôt de données (data warehouse, lakehouse)
Matérialisation des résultats	Résultats temporaires, transactions isolées	Agrégations stockées, cubes OLAP, résultats persistés

MS Acess vs SQLite vs DuckDB

Critère	Microsoft Access	SQLite	DuckDB
Type de base	Relationnelle avec interface graphique (OLTP)	Moteur SQL embarqué (OLTP)	Moteur analytique embarqué (OLAP)
Philosophie	Base de données bureautique, orientée utilisateur final	Base minimaliste intégrée aux applications	Moteur analytique local pour la data science
Fichier principal	`.accdb` ou `.mdb`	`.sqlite` ou `.db`	`.duckdb`
Architecture	Stockage ligne par ligne	Ligne par ligne (row-based)	Colonne par colonne (columnar)
Plateformes	Principalement Windows	Windows, macOS, Linux, Android, iOS	Windows, macOS, Linux
Serveur requis	Non (fichier local)	Non (serveurless)	Non (serveurless)
Interface utilisateur	Interface graphique complète (formulaires, rapports, macros VBA)	Aucune (CLI ou via code)	Aucune (CLI, Python, R, SQL)
Langages / API	VBA, ODBC, .NET	C, Python, Go, Java, etc.	Python, R, C++, Java, SQL
Type de requêtes optimisées	CRUD simples, formulaires, rapports	Petites requêtes transactionnelles	Requêtes analytiques (agrégations, joins lourds)
Performance (petit volume)	Bonne	Excellente	Très bonne
Performance (gros volume)	Faible (ralentit > 1 Go)	Moyenne (quelques Go max)	Excellente (plusieurs dizaines de Go)
Lecture/écriture concurrente	Très limitée	Une seule écriture à la fois	Multi-threaded, vectorisé
Gestion multi-utilisateurs	Possible en réseau local, peu fiable	Lecture multi, écriture unique	Lecture parallèle efficace
Indexation	Oui (B-Tree, primaire, secondaire)	Oui (B-Tree)	Pas nécessaire (optimisation vectorisée)
Transactions ACID	Oui	Oui	Oui
Support SQL standard	Partiel (SQL + fonctions Access)	Bon (SQL92 compatible)	Très bon (SQL moderne + fonctions analytiques)
Support des jointures complexes	Limité	Oui, basique	Oui, optimisé pour gros volumes
Gestion des types de données	Restreinte, orientée Access	Typage souple	Typage riche (compatibilité Arrow/Parquet)
Compatibilité Excel	Excellente (intégration native)	Possible via ODBC	Excellente (lecture directe de `.csv`, `.xlsx`, `.parquet`)
Import / Export de données	CSV, Excel, SQL via interface	SQL, CSV via scripts	CSV, Parquet, Arrow, JSON, Excel
Scripts / automatisation	Macros, VBA	API de langage (Python, C, etc.)	API Python/R ou SQL
Taille maximale de base	~2 Go	Jusqu'à plusieurs To selon système de fichiers	Plusieurs dizaines de Go (compressé, colonnaire)
Compression des données	Non	Non	Oui (compression colonnaire)
Format d'enregistrement	Propriétaire Microsoft	Binaire léger, ouvert	Colonnaire binaire optimisé

Critère	Microsoft Access	SQLite	DuckDB
Exécution dans un notebook (Jupyter, RStudio)	Non	Oui (avec modules)	Oui (intégration native)
Lecture de fichiers externes sans import	Non	Partiel (via extensions)	Oui (CSV, Parquet, Arrow, JSON)
Utilisation typique	Gestion interne, petites applis de bureau	Stockage local dans applis web/mobiles	Analyse de gros fichiers, prototypage analytique
Exemples d'usages	Gestion de stock, facturation simple, formulaires d'entreprise	Applications mobiles, sites web, IoT	Exploration de jeux de données, ETL local, data science
Compétences requises	Faible (bureautique)	Moyenne (développement)	Moyenne à avancée (data/SQL)
Licence	Propriétaire (Microsoft Office)	Open Source (Public Domain)	Open Source (MIT)
Communauté et support	Moyenne, orientée bureautique	Très large, développeurs	En forte croissance, orientée data
Portabilité du fichier	Moyenne (Windows)	Excellente	Excellente
Sécurité / chiffrement	Mot de passe basique	Extensions disponibles	Basique (pas encore complet)
Maintenance	Interface visuelle, peu flexible	Manuelle via scripts	Manuelle ou automatisée via code
Intégration cloud / moderne	Faible	Moyenne (via wrappers)	Bonne (compatible data lakes, formats modernes)

LICENSE

DUCK VBA DLL — GNU General Public License v3.0

yright (c) 2026 Etienne Lenoir

Ce projet (**DUCK VBA DLL**, incluant la DLL bridge C/C++, le toolkit VBA, les fichiers d'exemples et la documentation associée) est distribué sous la **GNU General Public License, version 3** (**GPL-3.0-only**).

Avis de licence (texte standard)

Ce programme est un logiciel libre : vous pouvez le redistribuer et/ou le modifier selon les termes de la **GNU General Public License** telle que publiée par la **Free Software Foundation**, **version 3** de la licence.

Ce programme est distribué dans l'espoir qu'il vous sera utile, mais **SANS AUCUNE GARANTIE** ; sans même la garantie implicite de **QUALITÉ MARCHANDE** ou d'**ADÉQUATION À UN USAGE PARTICULIER**.
Voir la **GNU General Public License** pour plus de détails.

Vous devriez avoir reçu une copie de la **GNU General Public License** avec ce programme. Si ce n'est pas le cas, consultez : https://www.gnu.org/licenses/gpl-3.0.html

Ce que cela implique (résumé pratique)

- **Usage libre**, y compris en contexte commercial.
- Si vous **redistribuez** (source ou binaire), vous devez :
  - fournir le **code source correspondant** (ou une offre équivalente conforme GPL),
  - conserver les **mentions** (copyright, licence),
  - redistribuer toute version modifiée sous **GPLv3** (copyleft).
- **Pas d'obligation** de publier vos modifications si vous ne redistribuez pas (usage interne).
- Pour un usage **SaaS / service réseau**, la GPLv3 ne force pas la publication du code (contrairement à l'AGPL).

  Note : la conformité exacte dépend de votre mode de distribution (classeurs, DLL, package, installateur, etc.). En cas de doute, relisez les termes GPLv3 et/ou faites valider par votre service juridique.

Composants tiers

Ce projet s'interface avec **DuckDB** (moteur de base de données), qui est un composant tiers sous sa propre licence (DuckDB est sous **MIT**).
La GPLv3 ci-dessus couvre **DUCK VBA DLL** ; vous devez également respecter les licences des composants tiers que vous distribuez (DuckDB, éventuelles extensions, etc.).

Marques / Logo / Identité visuelle (Trademark Policy)

Les noms **« DUCK VBA DLL »**, **« Duck-VBA »** (le cas échéant), ainsi que les logos, icônes et éléments d'identité visuelle associés (les **« Marques »**) sont des marques et/ou signes distinctifs appartenant à **Etienne Lenoir**.

Le logiciel est distribué sous **GNU General Public License v3.0 (GPLv3)**. **La GPLv3 ne concède aucun droit d'usage sur les Marques.** Toute utilisation des Marques doit respecter le droit des marques et la présente politique.

Utilisations autorisées (en général)

- **Usage nominatif** pour décrire ou référencer le logiciel (documentation, comparatifs, citations), à condition de ne pas suggérer une approbation, un partenariat ou une affiliation.
- Redistribution de **copies non modifiées** du logiciel en conservant les Marques **telles que fournies**.

Utilisations interdites sans autorisation écrite préalable

- Utiliser les Marques d'une manière qui **laisse entendre une approbation**, un parrainage ou une affiliation sans autorisation.
- Utiliser les Marques comme nom, logo ou branding principal d'une **version modifiée**, d'un fork ou d'une distribution dérivée, d'une manière susceptible de créer une confusion avec le projet officiel.
- **Modifier, adapter, recolorer, redessiner, animer, déformer** ou créer des versions dérivées des logos/icônes (sauf accord explicite).

Règle pratique pour les forks / versions modifiées

Vous pouvez modifier et redistribuer le code sous GPLv3, **mais** vous devez **renommer et/ou rebrander** votre distribution si nécessaire afin d'éviter toute confusion avec la distribution officielle.

Tout goodwill généré par l'usage autorisé des Marques bénéficie à leur propriétaire.

Réferences

1. DuckDB Contributors. DuckDB (site officiel). 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.