Application Shiny : gestionnaire de métadonnées SA_METADATA

Cette application Shiny, codée en R avec **shinydashboard**, est un gestionnaire de métadonnées pour la base `SA_METADATA`. Elle permet de se connecter à une base PostgreSQL, de créer et mettre à jour des tables de métadonnées (*Test_Info* et *Product_Info_Metadata*), et de détecter les tests et produits dont certaines données sont manquantes. L'interface est organisée en onglets (connexion, scan des tests, scan des produits, saisie manuelle, tables, import/export, debug), chacun offrant des fonctionnalités spécifiques.

Bibliothèques et configuration

Le script charge de nombreuses bibliothèques R : **shiny**, **shinydashboard** (UI), **DT** (tables interactives), **DBI** et **RPostgres** (connexion PostgreSQL), **readxl/writexl** (import/export Excel), **dplyr** (manipulations de données), **shinyWidgets/cssloaders/js** (UI améliorée), **fs/stringr** (fonctions utilitaires). Les valeurs statiques (listes déroulantes pour « GMPS TYPE », « COUNTRY CLIENT », etc.) sont définies en début de code. Les paramètres de connexion à la base sont codés dans `DB_CONFIG` (hôte, port, user, password, dbname) – la connexion est établie par `dbConnect(RPostgres::Postgres(), ...)` comme recommandé pour PostgreSQL[datacareer.derpostgres.r-dbi.org](datacareer.derpostgres.r-dbi.org), et fermée par `dbDisconnect`. La fonction `dbIsValid(con)` permet de vérifier la validité de la connexion avant chaque opération.

Création des tables de métadonnées

Deux tables principales sont utilisées :

- **Test_Info** : crée par la fonction `create_test_info_table()`. Elle contient les informations de tests (source_name, test_name, date, etc.) et des index pour optimiser les requêtes (index sur `source_name`, `test_name`, et un index unique sur la paire `(source_name, test_name)` pour empêcher les doublons).

- **Product_Info_Metadata** : crée par `create_product_info_metadata_table()`. Elle stocke les métadonnées produits (product_name, code_prod, base, ref, dosage) avec des index analogues sur `source_name` et `product_name`.

Chaque fonction de création exécute une requête SQL `CREATE TABLE IF NOT EXISTS` via `dbExecute`. Si les tables existent déjà, les fonctions retournent rapidement sans erreur.

Cette structure garantit que le schéma de métadonnées est mis en place avant toute lecture ou écriture.

Chargement et sauvegarde des données

Les fonctions `load_test_info_from_postgres(con)` et `load_product_info_metadata_from_postgres(con)` lisent les tables existantes en base (`dbReadTable(con, "Test_Info")`, etc.). Cela utilise **DBI**/**RPostgres** pour récupérer le contenu sous forme de *data.frame* en R. À l'inverse, les sauvegardes s'effectuent soit par `dbWriteTable()` (pour insérer de nouvelles lignes) soit par `dbExecute("UPDATE ...")` (pour mettre à jour). Par exemple, `save_test_info_to_postgres()` vérifie l'existence d'une entrée identifiée par `(source_name, test_name)` avec une requête `SELECT COUNT(*)`, puis met à jour ou insère selon le cas. Les fonctions SQL courantes sont utilisées : `dbGetQuery` pour requêtes SELECT, `dbExecute` pour UPDATE/CREATE, et `dbWriteTable` pour inserts. Notamment, `dbListTables(con)` permet de lister les tables disponiblesrpostgres.r-dbi.org, et la fonction `dbExistsTable(con, "Product_Info")` vérifie la présence de la table du script d'analyse. Comme rappelé dans la doc DBI, `dbWriteTable()` exécute plusieurs commandes SQL pour créer/écraser une table et y insérer les valeursrpostgres.r-dbi.org. De même, `dbReadTable(con, "mtcars")` peut récupérer le contenu d'une table existanterpostgres.r-dbi.org. Ces mécanismes DBI/RPostgres assurent la persistance des métadonnées.

Détection de données manquantes

Deux fonctions parcourent les données importées pour détecter les champs vides :

- **detect_missing_test_info(con)** : lit la table `Product_Info` générée par le script d'analyse (présumée contenir tous les tests), en extrait les noms de tests, puis compare à `Test_Info`. Si un test existe dans `Product_Info` mais qu'il manque dans `Test_Info`, il est marqué « À compléter ». Si un test est présent dans `Test_Info` mais certains champs (gmps_type, test_date…) sont vides ou `NULL`, il est aussi signalé. On utilise ici **dplyr** pour filtrer et manipuler les *data.frame*. Par exemple, `filter(is.null(gmps_type) | gmps_type == "")` repère les lignes incomplètes. Celles-ci sont affichées dans un tableau interactif.

- **detect_missing_product_info(con)** : analyse directement la table `Product_Info`. Elle filtre les produits dont des champs (`code_prod`, `base`, `ref`, `dosage`) sont vides ou NA, en indiquant aussi « À compléter ». Là encore, on applique un `filter` avec `is.null(...) | == ""` pour chaque champ.

Chaque fonction retourne un *data.frame* des éléments manquants, réinitialisé pour affichage (valeurs `NA` converties en chaînes vides). Un message informe du nombre d'éléments détectés, et des exemples sont loggués côté serveur. Ainsi, l'utilisateur peut facilement identifier quels tests ou produits nécessitent une saisie supplémentaire.

Interface utilisateur (UI) – shinydashboard

L'interface utilise **shinydashboard** pour créer un tableau de bord avec un en-tête (`dashboardHeader`), une barre latérale (`dashboardSidebar`) et un corps (`dashboardBody`). La barre latérale définit un `sidebarMenu` avec plusieurs `menuItem`, chacun pointant vers un onglet (`tabName`). Par exemple, l'item **« Scanner Test Info »** correspond au `tabName = "scan_tests"` et affiche le contenu de `tabItem(tabName = "scan_tests", ...)`. Il est crucial que chaque `menuItem` et `tabItem` partagent la même `tabName` pour que la navigation fonctionnerstudio.github.io. L'ordre des onglets reflète le menu :

- **Connexion SA_METADATA** (onglet `connection`) : bouton de connexion/déconnexion à la base, affiche l'état (`output$connection_status`) et les tables disponibles.

- **Scanner Test Info** (`scan_tests`) : bouton de scan et tableau des tests manquants (via `DT::dataTableOutput`).

- **Scanner Product Info** (`scan_products`) : similaire, pour les produits.

- **Saisie Test Info** (`manual_test`) : formulaire manuel de saisie d'un test. Comprend des `selectInput` ou `textInput` pour chaque champ (source_name, test_name, date, etc.), avec validation en temps réel. Deux boutons permettent d'enregistrer ou d'enregistrer+suivant.

- **Saisie Product Info** (`manual_product`) : formulaire pour renseigner un produit manquant (`code_prod`, `base`, `ref`, `dosage`).

- **Tables SA_METADATA** (`postgres_tables`) : affiche les tables `Test_Info` et `Product_Info` en base dans deux `DT::datatable` séparés, avec bouton « Actualiser ».

- **Import/Export** (`import`) : prévu pour l'import/export Excel (actuellement en développement).

- **Debug** (`debug`) : affiche des diagnostics (état de connexion, tables existantes, statuts de données).

Les boîtes `box()` et `tabItems()` structurant l'UI décrivent les contenus textuels et les sorties. Des alertes colorées (bootstrap) guident l'utilisateur. Le design suit les bonnes pratiques Shiny : on a un `sidebarMenu` avec des `tabName` correspondants à des `tabItem` dans le `dashboardBody`rstudio.github.io. Les `actionButton` déclenchent des `observeEvent` côté serveur pour réaliser les opérations.

Logique serveur (Server)

La partie serveur définit la logique réactive et les gestionnaires d'événements :

- **Connexion à PostgreSQL** : lors du clic sur « Se Connecter », la fonction `create_postgres_connection()` ouvre la connexion via `dbConnect(RPostgres::Postgres(), ...)`. Si elle réussit, un `reactiveVal` `postgres_con` est mis à jour et l'interface indique la connexion établie (texte vert). De même, le bouton « Se Déconnecter » ferme la connexion (`dbDisconnect`), remet le statut à faux, et reset l'interface. On stocke le statut de connexion dans `connection_status` (reactiveVal booléen).

- **Mises à jour des listes dynamiques** : après connexion, on récupère via la base les listes dynamiques nécessaires au formulaire, en exécutant des requêtes (par ex. `get_unique_product_names(con)` lit la table `Product_Info` pour en extraire les noms uniques). Ces listes alimentent les `selectizeInput` (« Nom Produit », « Source Name ») de manière réactive.

- **Validation des champs** : en temps réel, des `observeEvent` sur les champs `input$test_date`, `input$sc_request`, `input$dosage_input` appliquent des regex pour vérifier les formats (ex: `validate_date_format` impose `DD/MM/YYYY`). Un code CSS (`.validation-error`, `.validation-success`) change la bordure des champs et des messages d'alerte sont affichés en cas d'erreur. C'est le principe de feedback immédiat courant en Shiny.

- **Scan des tests et produits manquants** : lorsque l'utilisateur clique sur « Scanner Test Info Manquants », l'observeur correspondant (`observeEvent(input$scan_test_info_btn)`) appelle `detect_missing_test_info(con)`. Le *data.frame* résultant est stocké dans `missing_test_info` (reactiveVal) et affiché dans `output$missing_test_info_table` via `DT::renderDataTable`. Un statut réactif `test_scan_completed` permet de conditionner l'affichage du tableau. Un mécanisme similaire gère le scan des produits manquants. Ce pattern (lancer l'opération, remplir une reactiveVal puis rafraîchir l'UI) est typique des applications Shiny.

- **Interactions de double-clic** : dans les tableaux DT des tests/produits manquants, un double-clic sur une ligne déclenche un `input$..._cell_clicked` contenant

l'index de la ligne. Un observeur récupère alors la ligne sélectionnée et pré-remplit le formulaire manuel correspondant (`updateSelectizeInput`, `updateTextInput`, etc.). Puis il bascule automatiquement vers l'onglet de saisie (via `updateTabItems`). Ce mécanisme permet à l'utilisateur de cliquer dans le tableau pour charger un test/produit en mode édition, ce qui améliore l'ergonomie.

- **Sauvegarde des saisies** : les boutons **Enregistrer** du formulaire test ou produit déclenchent un observeur qui collecte les valeurs `input$...` dans un *data.frame*, les valide (champs obligatoires, formats) puis appelle `save_test_info_to_postgres()` ou `save_product_info_to_postgres()`. Ces fonctions effectuent un INSERT ou UPDATE en base. En cas de succès, on notifie l'utilisateur et on relance le scan des éléments manquants pour mettre à jour la liste. Les versions *« Et Suivant »* enchaînent l'enregistrement puis chargent automatiquement le prochain élément manquant (index 1 du tableau).

- **Affichage des tables et diagnostics** : deux `DT::renderDataTable` affichent le contenu actuel de `Test_Info` et de `Product_Info` (cette dernière crée par le script d'analyse). Un bouton « Actualiser » permet de rafraîchir manuellement l'affichage. L'onglet **Debug** affiche en texte brut l'état de la connexion, les tables listées (`dbListTables(con)`) et quelques statistiques (nombre de lignes dans chaque table, nombre de tests/prod. manquants). Les reactiveVals `missing_test_info()` et `missing_product_info()` sont utilisées pour ces calculs. Enfin, la fonction `session$onSessionEnded` se charge de fermer proprement la connexion à la base à la fin de la session.

Dans l'ensemble, la logique serveur utilise abondamment **reactiveVal** pour stocker l'état (connexion, liste manquante, statut de scan). Comme l'explique la documentation Shiny, un objet `reactiveVal` agit comme une variable *réactive* : la lire crée une dépendance, et la modifier notifie les réactifs dépendantsshiny.posit.co. Les autres briques réactives classiques sont utilisées (observeEvent, renderText, renderDataTable, reactive expressions) pour relier les événements UI aux opérations en base et à la mise à jour de l'interface.

Intégration PostgreSQL et opérations DBI

L'application communique avec PostgreSQL via le package **RPostgres** (s'appuyant sur DBI). Le code illustre plusieurs cas typiques :

- **Connexion** : `dbConnect(RPostgres::Postgres(), host=..., port=..., user=..., password=..., dbname=...)` ouvre une session. Par exemple, on peut écrire `con <- dbConnect(RPostgres::Postgres(), dbname = db, host = host_db, port = db_port, user = db_user, password = db_password)`datacareer.de. La fonction `dbIsValid(con)` vérifie si la connexion est toujours active avant de l'utiliser.

- **Listage des tables** : `dbListTables(con)` renvoie tous les noms de tables de la base connectée. Cet appel est utilisé pour afficher les tables disponibles dans l'UI et dans le diagnosticrpostgres.r-dbi.org.

- **Lecture d'une table** : `dbReadTable(con, "Test_Info")` récupère les données sous forme de data.frame. Par exemple, pour afficher la table `mtcars` temporaire (dans l'exemple RPostgres) on utiliserait `dbReadTable(con, "mtcars")`rpostgres.r-dbi.org. Dans notre code, cette fonction sert à charger `Test_Info` existant ou `Product_Info`.

- **Écriture/MAJ de données** : plusieurs méthodes sont employées. Pour insérer de nouvelles lignes, `dbWriteTable(con, "Test_Info", test_data, append=TRUE)` écrase pas la table existante mais ajoute (avec `row.names=FALSE`). D'après la doc, `dbWriteTable()` crée/écrase la table et y insère toutes les données du *data.frame*rpostgres.r-dbi.org. Le code utilise aussi `dbExecute(con, SQL, params)` pour mettre à jour des lignes existantes (requête `UPDATE Test_Info SET ... WHERE source_name = $1 AND test_name = $2`). Ces appels SQL paramétrés permettent d'éviter les injections et d'actualiser seulement les champs modifiés. Après chaque insertion ou mise à jour réussie, on affiche une notification à l'utilisateur. Les exemples officiels montrent cet usage de DBI : par exemple, on peut exécuter `dbGetQuery(con, "SELECT COUNT(*) FROM Test_Info")` pour obtenir le nombre de lignes, ou faire un `dbWriteTable(con, "newtable", dataframe)` pour importer des donnéesrpostgres.r-dbi.orgrpostgres.r-dbi.org.

- **Vérifications existantes** : avant insertion, une requête `SELECT COUNT(*) FROM Test_Info WHERE source_name = ...` assure l'unicité. De plus, `dbExistsTable(con, "Product_Info")` renvoie TRUE/FALSE selon que la table existe, ce qui permet de gérer les cas où le script d'analyse génère ou non la table. Les indices uniques (créés au moment des `CREATE TABLE` via `CREATE UNIQUE INDEX`) empêchent les doublons à deux niveaux.

En résumé, le code applique les patterns habituels de connexion et manipulation SQL en R : établir la connexion, utiliser `dbListTables`, `dbExistsTable`, `dbReadTable`, `dbWriteTable`, `dbExecute`/`dbGetQuery`, puis refermer la connexion. Cette approche est recommandée pour les applications Shiny en production, où les données sont stockées en base plutôt que dans des CSV locauxdatacareer.derpostgres.r-dbi.org.

Organisation et bonnes pratiques Shiny

Enfin, quelques remarques générales :

- **Structure réactive** : L'usage de `reactiveVal` (par ex. `postgres_con <- reactiveVal(NULL)`) permet de partager la connexion et d'autres états entre observateurs sans globaliser de variables. Les `observeEvent` déclenchent des actions uniquement lors d'événements (clics de bouton, changements d'input)shiny.posit.co. Cette programmation événementielle est essentielle dans Shiny : on attend qu'un utilisateur clique sur « Scannez » ou « Enregistrer » pour lancer le code correspondant.

- **Navigation par onglets** : shinydashboard facilite la création d'onglets comme dans l'exemple structurel. Chaque `menuItem` du `sidebarMenu` a un `tabName` qui correspond au `tabItem` du corpsrstudio.github.io. Dans le code, les fonctions `updateTabItems(session, "sidebarMenu", "manual_test")` (et similaires) changent dynamiquement l'onglet actif, ce qui permet de guider l'utilisateur vers le formulaire de saisie après un double-clic sur un test manquant.

- **Retour visuel et notifications** : l'interface utilise `showNotification()` pour informer l'utilisateur des états (succès, erreur, avertissement). Les `withSpinner()` autour des `datatable` affichent un chargement animé tant que l'opération est en cours. Les classes CSS personnalisées (`.validation-error`, `.status-connected`, etc.) améliorent l'UX en colorant les éléments selon leur statut.

En combinant ces éléments – accès base de données, UI structurée par `shinydashboard`, logique réactive – l'application propose un outil complet pour gérer les métadonnées, tout en se conformant aux bonnes pratiques de Shiny (séparation UI/serveur, réactivité maîtrisée, encapsulation de la connexion)rstudio.github.iodatacareer.de.

Sources : documentation RPostgres/DBI pour la connexion et manipulation SQLdatacareer.derpostgres.r-dbi.org ; exemples et références Shiny pour reactiveVal et structure shinydashboardrstudio.github.ioshiny.posit.co

Documentation Technique du Projet *Analyse Senso* (2025, multi-bases et tracking)

Le script **Analyse Senso** est une application R complète destinée à traiter et analyser des données expérimentales sensorielles. Il intègre notamment :

- **Multi-bases PostgreSQL** : Sauvegarde des données brutes, des résultats et du suivi des juges dans plusieurs bases de données distinctes (`SA_RAW_DATA`, `SA_RESULTS_DATA`, `SA_JUDGES`, `SA_METADATA`).

- **Système de tracking** des fichiers traités : Pour éviter la relecture de fichiers déjà analysés, en enregistrant un hash MD5, le statut de traitement et les métadonnées.

- **Analyses statistiques** automatiques : Détection du type de test (force, proximité, triangulaire, etc.), validation des données, retrait itératif des juges aberrants, tests ANOVA et post-hoc (SNK ou Duncan) pour les différences entre produits, tests binomiaux pour les tests triangulaires.

- **Interface de reporting** : Génération de fichiers Excel individuels pour chaque source de données et d'un rapport consolidé global avec synthèses et statistiques.

- **Tables auxiliaires pour application Shiny** : Création et sauvegarde d'informations produits et tests à compléter manuellement.

Ce document détaille l'architecture, les fonctions clés et le flux de traitement de ce script.

Chargement des bibliothèques et configuration

- Le script commence par charger les librairies R essentielles : **tidyverse** (dplyr, tidyr, etc.), **readxl**, **agricolae** (pour les tests SNK/Duncan), **DBI**, **RPostgreSQL** et **RPostgres** (pour la connexion aux bases PostgreSQL), **odbc**, **fs**, **writexl**, **stringr** et **digest** (pour calculer les hash MD5)【source】.

- La configuration des bases de données est définie via `DB_CONFIG` (hôte, port, utilisateur, mot de passe) et `DATABASES` (noms logiques des quatre bases SQL utilisées : `SA_RAW_DATA`, `SA_RESULTS_DATA`, `SA_JUDGES`, `SA_METADATA`).

- Une fonction `debug_database_connections()` teste la connexion au serveur PostgreSQL et la présence de ces bases sur le serveur. Elle affiche la liste des bases existantes et tente de lister les tables des bases requises, facilitant le

diagnostic initial de l'environnement.

Connexion multi-bases et sécurité

- **Fonction `create_db_connection(database_name)`** : Tente d'établir une connexion à une base PostgreSQL spécifiée via `dbConnect(RPostgres::Postgres(), ...)`. En cas d'erreur, elle renvoie `NULL`. Elle affiche un message en console sur la réussite ou l'échec de la connexion.

- **Fonction `safe_disconnect(con)`** : Déconnecte proprement une connexion si elle est valide, pour éviter les connexions orphelines.

- Ces fonctions sont utilisées dans tout le script pour ouvrir et fermer les connexions avant/après chaque opération sur chaque base.

Création des tables dans les bases de données

Le script définit plusieurs fonctions pour créer les tables nécessaires, chacune vérifiant d'abord l'existence de la table :

- **`create_raw_data_table(con)`** (base `SA_RAW_DATA`) : Crée la table `rawdata` contenant les données brutes du fichier (avec des colonnes `id`, `source_name`, `trial_name`, `cj`, `product_name`, `attribute_name`, `nom_fonction`, `value`, `judge_status`, timestamps). Des index sont ajoutés sur `source_name`, `trial_name` et `product_name`.

- **`create_results_table(con)`** (base `SA_RESULTS_DATA`) : Crée la table `test_results` pour tous types de résultats de tests. Colonnes : `id`, `source_name`, `idtest`, `test_type`, `segment`, `segment_id`, `product_name`, `classe`, statistiques (`mean_value`, `sd_value`, `n_observations`), indicateurs ANOVA à 5% et 10%, plus les champs spécifiques aux tests triangulaires (`reference`, `candidate`, `n_total`, `n_correct`, `p_value`, `decision`). Des index sont placés sur `source_name`, `test_type` et `product_name`.

- **`create_judges_table(con)`** (base `SA_JUDGES`) : Crée la table `judge_tracking` qui enregistre le suivi des juges. Colonnes : `id`, `source_name`, `cj`, nombre d'évaluations (`nb_evaluations`), `moyenne_score`, nombre d'attributs évalués, nombre de produits évalués, totaux de tests, tests conservés, taux de conservation, `judge_status`, dates. Index sur `source_name` et `cj`.

- **`create_metadata_table(con)`** (base `SA_METADATA`) : Crée la table `databrute` (data brute metadata), avec colonnes `id`, `idtest`, `productname`, `sourcefile`,

timestamps, et un index unique (`sourcefile, productname`). Cette table conserve les couples fichier-produit ingérés.

En plus, pour l'application Shiny, deux autres tables sont créées dans `SA_METADATA` :

- **`create_product_info_table(con)`** : Table `Product_Info` pour lister les produits uniques d'un fichier (colonnes : `id`, `source_name`, `product_name`, `idtest`, et champs vides pour `code_prod`, `base`, `ref`, `dosage` à remplir par l'utilisateur via l'app).
- **`create_test_info_table(con)`** : Table `Test_Info` pour les tests (colonnes : `id`, `source_name`, `test_name`, et champs vides pour divers attributs du test comme `gmps_type`, `gpms_code`, `type_of_test`, etc., à remplir manuellement).

Chaque fonction de création de table gère les éventuelles erreurs et affiche des messages de succès ou d'erreur dans la console.

Sauvegarde des données dans les bases

Le script propose des fonctions pour enregistrer les données extraites dans les bases correspondantes :

- **`save_raw_data_to_db(raw_data, source_name)`** :
  - Ouvre la connexion à `SA_RAW_DATA` et crée la table `rawdata` si nécessaire.
  - Transforme le dataframe `raw_data` (issu de la lecture Excel) en format de la table (`source_name`, `trial_name`, `cj`, `product_name`, `attribute_name`, `nom_fonction`, `value`, `judge_status="conserved"`).
  - Supprime d'abord les enregistrements existants pour ce `source_name` (via `DELETE FROM rawdata WHERE source_name = $1`).
  - Insère (`dbWriteTable`) les nouvelles données.
  - Affiche un message indiquant le nombre de lignes sauvegardées.
- **`save_results_to_db(results_data, source_name, test_type)`** :
  - Connexion à `SA_RESULTS_DATA`, création de la table `test_results` si nécessaire.

    - Selon `test_type`, structure différemment le dataframe `results_data` :
        - Pour `Triangular`, on conserve seulement `idtest`, `reference`, `candidate`, `n_total`, `n_correct`, `p_value`, `decision`.
        - Sinon (Strength, Proximity, Malodour), on conserve `idtest`, `segment`, `segment_id`, `product_name`, `classe`, `mean_value`, `sd_value`, `n_observations`, et indicateurs ANOVA.
    - Supprime les résultats existants pour le fichier (`DELETE ... WHERE source_name`), puis insère les nouveaux.
    - Affiche un message de confirmation.
- **`save_judges_to_db(judge_data, source_name)`** :
    - Connexion à `SA_JUDGES`, création table `judge_tracking` si besoin.
    - Prépare `judge_data` (suivi des juges par fichier) en ajoutant `source_name`, renommant la colonne `CJ`, calculant les champs (`nb_evaluations`, `moyenne_score`, etc.) à partir des colonnes présentes, et en fixant `judge_status` et `date_analyse`.
    - Supprime anciens enregistrements (`DELETE WHERE source_name`), puis insère les nouveaux.
    - Confirme le nombre de lignes sauvées.
- **`save_product_info_complete(raw_data, source_name)`** et **`save_test_info_complete(raw_data, source_name)`** :
    - Ces fonctions sauvegardent dans `SA_METADATA` les enregistrements *vides* pour chaque produit et chaque test du fichier.
    - Elles créent les tables `Product_Info` et `Test_Info` si nécessaire.
    - Pour `Product_Info`, on extrait chaque couple unique `(TrialName, ProductName)` du dataframe brut, on crée un enregistrement avec `source_name`, `product_name`, `idtest=TrialName`, et les autres champs vides.
    - Pour `Test_Info`, on extrait chaque `TrialName` unique comme `test_name`, créant un enregistrement avec `source_name`, `test_name` et les autres attributs vides. La fonction vérifie aussi qu'un enregistrement identique

n’existe pas déjà avant d’insérer.

- Utile pour préparer les informations de configuration des produits/tests dans l’interface Shiny après coup.

Détermination du type de test

- La fonction **`determine_test_type(segments)`** prend la liste de segments (sous-ensembles de données par `AttributeName` et `NomFonction`) et analyse leur contenu pour identifier le type général de test du fichier :
  - Si un segment a `NomFonction` contenant “Triangulaire” ou “triangle”, on retourne `"Triangular"`.
  - Sinon, si un segment a `AttributeName` contenant “prox” (proximité), on retourne `"Proximity"`.
  - Sinon, si un segment a `AttributeName` contenant “odeur corporell” (cas de Malodour), on retourne `"Strength with Malodour"`.
  - Sinon, on renvoie `"Strength"` (test de force par défaut).
- Ce type de test global déterminera la méthode d’analyse appliquée aux segments.

Système de tracking des fichiers traités

- Un fichier Excel `TRACKING_FICHIERS.xlsx` est utilisé pour enregistrer le suivi de chaque fichier traité. Les colonnes sont : `Fichier`, `Chemin_Complet`, `Hash_MD5`, `Date_Traitement`, `Statut`, `Taille_Fichier`, `Nb_Lignes_Results`.
- **`load_tracking_data()`** : Charge ce fichier s’il existe, sinon initialise un tibble vide avec la structure appropriée.
- **`calculate_file_hash(file_path)`** : Calcule le hash MD5 du fichier donné (pour détecter les doublons/versions inchangées).
- **`is_file_already_processed(file_path, tracking_data)`** : Vérifie dans le tableau de tracking si ce fichier (nom + hash) existe déjà avec `Statut=="SUCCES"`. Utile pour sauter les fichiers déjà traités.
- **`update_tracking(file_path, statut, nb_lignes, tracking_data)`** : Ajoute ou remplace l’entrée correspondant au fichier dans `tracking_data` avec le

nouveau statut (`"SUCCES"`, ou code d'erreur) et d'autres métadonnées (date, nb lignes, taille, hash).

- **`save_tracking_data(tracking_data)`** : Écrit le tibble en Excel (`TRACKING_FICHIERS.xlsx`) en fin de traitement pour conserver l'historique.

Ces fonctions permettent d'**éviter le retraitement** de fichiers et de garder un journal complet de l'analyse.

Fonctions utilitaires et de validation

- **`log_probleme(type, details, fichier)`** : Ajoute un message d'erreur ou d'avertissement lié aux données (par exemple format incorrect, valeurs aberrantes, segments manquants, etc.) dans une liste globale `data_issues_log`, avec un préfixe indiquant le type de problème.

- **`validate_data_consistency(file_data)`** : Vérifie que la colonne `Value` du dataframe ne contient pas de valeurs non numériques (zéros, lettres, etc.). Renvoie une liste d'issues détectées (par ex. nombre de valeurs non numériques). Le cas échéant, ces anomalies sont loguées par `log_probleme`.

Ces contrôles facilitent le repérage de données potentiellement erronées avant l'analyse.

Tracking des juges (création de table)

- **`create_judge_tracking_table(all_judge_info, all_raw_data)`** : À partir de deux dataframes agrégés globalement sur tous les fichiers :
    - `all_raw_data` (les données brutes de toutes sources, avec colonnes `SourceFile`, `CJ`, `Value`, `AttributeName`, `ProductName`).
    - `all_judge_info` (résumé de tous les juges retirés dans chaque fichier, colonnes `File`, `Segment`, `RemovedJudges`).
    - Elle calcule, pour chaque juge (`CJ` par fichier), le nombre d'évaluations (`NbEvaluations`), score moyen (`MoyenneScore`), nombre d'attributs et produits évalués, et la date d'analyse.
    - Si une liste `RemovedJudges` existe, elle marque ces juges comme `"removed"`, sinon `"conserved"`.

- Enfin, elle calcule pour chaque juge le nombre total de tests auxquels il a participé (`NbTestsTotal`), le nombre de tests conservés (`NbTestsConserve`), et le taux de conservation (`TauxConservation`).

- Retourne un tibble `judge_tracking` consolidant ces informations. Utile pour récapituler le comportement des juges sur l'ensemble des analyses.

Gestion spécifique des tests de proximité

- **`handle_proximity_test(segment)`** : Appliqué sur un segment de test de proximité (attribut lié à la proximité sensorielle) :

  - Détermine le produit de référence : celui avec la moyenne la plus faible (`bench_product`).

  - Identifie les juges à filtrer : ceux dont le score sur le produit de référence est >4 ou dont un score sur un autre produit est ≤ `bench_score - 1` (critère spécifique de détection de juge incohérent pour la proximité).

  - Renvoie une liste contenant : le segment filtré sans ces juges (`segment`), les juges retirés (`removed_judges`), le nombre de juges initial et final.

Traitement des segments triangulaires

- **`process_triangular_segments(segments, file_path, file_test_type)`** : Pour les tests triangulaires, où plusieurs segments « triangulaires » doivent être fusionnés :

  - Identifie tous les segments dont `NomFonction` indique un test triangulaire.

  - Si aucun, retourne `NULL`.

  - Sinon, concatène toutes ces données en un seul tableau (`all_triangular_data`).

  - Calcule `n_total` (nombre total de juges) et `n_correct` (nombre de juges ayant identifié correctement le produit cible, supposé codé `Value==1`).

  - Effectue un test binomial (`binom.test`) avec p=1/3 (chance aléatoire en test triangulaire) pour obtenir une **p-value**.

    - Décide « Significatif » si $p<0.05$, sinon « Non significatif ».
    - Définit un produit de référence (premier produit observé) et un candidat (deuxième).
    - Construit un tibble de résultat avec colonnes `IDTEST`, `REFERENCE`, `CANDIDATE`, `N`, `CORRECT`, `P_VALUE`, `DECISION`, `TESTTYPE`.
    - Retourne ce résultat qui sera inséré dans `test_results`.

Analyse itérative des juges aberrants

- **`analyze_judges_iterative(segment)`** : Filtre itérativement les juges d'un segment standard (non triangulaire) en fonction de l'effet juge :
    - Si le segment correspond à un test Malodour (odeur corporelle), aucun filtrage n'est fait (on retourne les juges tels quels).
    - Sinon, on répète :
        - Réaliser une ANOVA (`aov(Value ~ CJ)`) sur les notes par juge.
        - Si p-value de l'effet juge ≥ 0.05, arrêter (aucun effet significatif).
        - Si le nombre de juges actuels ≤ 8, ou qu'on a déjà supprimé jusqu'à 2/3 des juges initiaux, arrêter (seuil minimal de juges atteints).
        - Sinon, calculer la moyenne de chaque juge, puis la déviation absolue par rapport à la moyenne globale.
        - Retirer le juge avec la plus grande déviation (suivant la méthode de détection de juge aberrant).
        - Recalcule avec le juge retiré et répéter.
    - Accumule la liste des juges retirés et retourne les juges restants, le nombre initial et final, etc.
    - Cette approche enlève les juges dont les notations sont systématiquement trop éloignées du consensus.

Analyse des produits (tests statistiques)

- **`analyze_products(segment, segment_index, file_path, file_test_type)`** : Génère les résultats pour un segment et un type de test donné :
    - Si `file_test_type` est **Triangular**, appelle la procédure du test triangulaire (calcul de p-value binomiale, décision, référence/candidat) et renvoie un résultat avec colonnes (`IDTEST`, `REFERENCE`, `CANDIDATE`, `N`, `CORRECT`, `P_VALUE`, `DECISION`, `TESTTYPE`).
    - Sinon (Strength, Malodour, Proximity) :
        - Vérifie qu'il y a au moins 2 produits dans le segment ; sinon renvoie `NULL`.
        - Calcule les statistiques de base par produit : moyenne (`Mean`), écart-type (`Sd`), effectif (`n`).
        - Réalise une ANOVA (`aov(Value ~ ProductName)`) pour tester l'effet produit. Retient *p*-value (`p_value_produit`).
        - Détermine deux indicateurs logiques : `anova_5pct` (<5%) et `anova_10pct` (<10%).
        - Si l'effet produit est significatif à 10% (`anova_10pct == TRUE`), applique un test post-hoc de Student-Newman-Keuls (SNK) pour classer les produits en groupes (*letters*). En cas d'erreur SNK, utilise Duncan en alternative.
        - Sinon, attribue à tous les produits le même groupe « a ».
        - Assemble un dataframe `result_df` avec les produits, leur *classe* (groupement par test post-hoc), leurs moyennes, écarts, effectifs.
        - Ajoute les colonnes communes en fonction du type :
            - Pour **Strength** et **Strength with Malodour** : on produit un tibble avec `IDTEST`, `SEGMENT` (concaténation d'attribut et fonction), `IDSEGMENT`, `PRODUCT`, `CLASSE`, `MEAN`, `SD`, `N`, `TESTTYPE`, `ANOVA à 5%`, `ANOVA à 10%`.
            - Pour **Proximity** : similaire mais sans colonnes ANOVA (ici on considère seulement classement des produits).
        - Retourne ce tibble de résultats par segment.

- Note : Les noms de colonnes comme `IDSEGMENT`, `PRODUCT`, etc., sont alignés sur la table `test_results`.

Vérification pré-analyse des segments

- **`verify_segments(segment)`** : Pour chaque segment (groupe d'attributs/fonctions), cette fonction contrôle :
  - Le nombre de produits distincts, juges distincts, valeurs manquantes.
  - Pour les tests non triangulaires : comptes de valeurs non numériques, supérieures à 10, négatives, etc.
  - Vérifie les seuils minimaux : au moins 3 juges (`MinJudgesOK`), au moins 2 produits (`MinProductsOK`).
  - Retourne une liste avec ces indicateurs (pour repérer des segments problématiques avant analyse). Les anomalies (p.ex. trop peu de juges ou produits, valeurs hors limites) sont ensuite enregistrées via `log_probleme`.

Boucle principale d'analyse

1. **Chargement du tracking et des fichiers** : On charge le suivi existant (`tracking_data`) et on liste tous les fichiers Excel (`.xlsx`) dans le dossier source (`raw_data_dir`).
2. **Parcours des fichiers** : Pour chaque fichier détecté :
   - **Vérification du traitement préalable** : Si le fichier (par nom et hash) figure déjà en succès dans le tracking, on le *skippe*.
   - Sinon, on procède à un nouveau traitement :
     1. Lecture de la feuille **`Results`** du fichier Excel. S'il manque ou erreur, on logge le problème et marque le fichier en échec dans le tracking.
     2. Validation : s'assure qu'il n'y a qu'un seul `TrialName` dans le fichier (sinon on interrompt le traitement), et signale les problèmes de cohérence via `validate_data_consistency`.
     3. Transformation : convertit la colonne `Value` en numérique, standardise `JudgeStatus="conserved"`. Stocke les données

brutes dans `all_raw_data`.

4. **Segmentation** : on découpe les données par combinaison `AttributeName` + `NomFonction` (`group_split()`), créant plusieurs segments à analyser séparément.

5. **Type de test** : on détermine le type global du fichier (`determine_test_type`).

6. **Vérification des segments** : on applique `verify_segments` à chacun et on logge les problèmes (trop peu de juges, de produits, valeurs hors plage).

- **Traitement adaptatif des segments** :

  1. Pour chaque segment valide (assez de juges/produits) :

     - Si le test est **Triangular** : on concatène simplement les segments sans filtrer de juges (`process_triangular_segments` gère la fusion après).

     - Si **Proximity** : on utilise `handle_proximity_test` pour filtrer les juges incohérents dans ce segment.

     - Sinon (Strength, Malodour) : on applique `analyze_judges_iterative` pour retirer les juges aberrants itérativement.

  2. On conserve le segment filtré (`segment$segment`) et on mémorise les juges supprimés (pour la génération de logs et suivi des juges).

- **Consolidation des données traitées** :

  1. On regroupe tous les segments traités en un seul dataframe `final_data`.

  2. On traite les segments triangulaires groupés via `process_triangular_segments`. On traite les autres segments avec `analyze_products` pour chaque segment, collectant les résultats statistiques.

  3. Le **résultat d'analyse** du fichier (`file_results`) est la combinaison des résultats triangulaires (s'il y a lieu) et des résultats standard de `analyze_products`. Ce tibble est stocké dans `all_results`.

- **Tracking des juges pour le fichier** :

    1. On construit un petit tableau `judge_tracking_table` pour ce fichier, en appelant `create_judge_tracking_table` sur les informations des juges retirés et toutes les données brutes du fichier. Ce tableau contient pour chaque juge du fichier le suivi détaillé (conserve/enlève, taux, etc.).

- **Sauvegarde en bases de données** :

    1. Appel à `save_raw_data_to_db` avec les données brutes lues (`file_data`) pour enregistrer dans `SA_RAW_DATA`.

    2. Si `file_results` n'est pas vide, appel à `save_results_to_db(file_results, source_name, file_test_type)` pour `SA_RESULTS_DATA`.

    3. Si un tracking juges existe, appel à `save_judges_to_db(judge_tracking_table, source_name)` pour `SA_JUDGES`.

    4. `save_product_info_complete(file_data, source_name)` et `save_test_info_complete(file_data, source_name)` pour insérer dans `SA_METADATA` les enregistrements de produits/tests (champs vides).

- **Mise à jour du tracking** : On marque le fichier comme traité avec succès (`Statut="SUCCES"` et nombre de lignes) dans le fichier de suivi `tracking_data`.

3. **Production des fichiers Excel individuels** : Pour chaque fichier traité avec succès, on génère un rapport Excel (`ANALYSE_<nom>.xlsx`) contenant plusieurs feuilles :

    - **Données_Brutes** : les données initiales du fichier avec la colonne `JudgeStatus` mise à jour (juges retirés marqués "removed").

    - **Resultats_Analyse** : le tableau `file_results` obtenu (ou un message d'absence de résultat si vide).

    - **Tracking_Juges** : la table de suivi des juges (`judge_tracking_table`) pour ce fichier.

    - **Juges_Retires** : liste des juges retirés par segment (si applicables).

- **Resume_Fichier** : résumé meta du traitement (nom du fichier, trial, date de traitement, nombre de lignes/ségments/juges/produits, type de test, statut).
- **Problemes_Detectes** : liste des problèmes loggés spécifiques au fichier.
- Ce fichier de rapport est sauvegardé dans le dossier `output_base_dir`.

4. **Génération du rapport consolidé global** : Après avoir traité tous les fichiers, on crée un fichier unique `ANALYSE_CONSOLIDEE_GLOBALE.xlsx` rassemblant :
    - **Resume_Global** : statistiques générales (nombre de fichiers trouvés, traités, réussis/échoués, taux de succès, dossiers source/sortie).
    - **Tous_Resultats** : concaténation de tous les résultats analytiques individuels (`all_results`).
    - **Stats_Types_Tests** : regroupement par type de test du nombre de tests et fichiers concernés.
    - **Stats_Produits** : pour les tests non triangulaires, statistiques par produit (nombre d'occurrences, moyennes).
    - **Tests_Triangulaires** : tableau des seuls résultats triangulaires (référence, candidat, p-values, décision).
    - **Tracking_Juges_Global** : agrégation de tous les juges retirés dans tous les fichiers (information file/segment/juges).
    - **Stats_Juges_Retires** : résumé du nombre de retraits par juge sur tous les fichiers.
    - **Stats_Donnees_Brutes** : statistiques sur les données brutes globales (par fichier : nb lignes, nb juges/produits/attributs, min/max des valeurs, etc.).
    - **Problemes_Detectes** : liste de tous les problèmes détectés sur tous les fichiers, classés par type (onglet manquant, incohérence, etc.).
    - **Resume_Problemes** : récapitulatif du nombre d'occurrences de chaque type de problème.
    - **Tracking_Fichiers** : version finale du tableau de suivi (fichiers et leur statut).
    - Ce rapport global fournit une vue d'ensemble de l'analyse effectuée.
5. **Nettoyage et conclusion** :

    - À la fin, le script ferme les connexions, nettoie l'environnement et imprime un résumé de fin avec le nombre de fichiers traités avec succès, en erreur, ainsi que la liste des fichiers et résultats générés. Des messages indiquent notamment que les données ont été sauvegardées dans chaque base et la localisation des fichiers Excel générés.

En résumé

Le script **Analyse Senso (2025)** implémente une chaîne complète de traitement des données sensorielles :

- **Préparation** : lecture des fichiers sources, validation et transformation des données.
- **Analyse statistique** : classification du type de test, filtrage des juges, tests ANOVA ou binomiaux, calcul des moyennes et des groupes de produits.
- **Persistance** : stockage des données brutes, résultats, métadonnées et suivi des juges dans des bases PostgreSQL, permettant un accès centralisé.
- **Suivi de version** : enregistrement des fichiers traités avec leurs états pour éviter la duplication et faciliter le débogage.
- **Reporting** : génération de rapports détaillés, tant individuels que consolidés.

Chaque étape clé est confiée à une fonction dédiée, avec gestion des erreurs et des messages explicites pour faciliter la maintenance. Cette architecture modulaire facilite l'évolution du script (ajout de nouveaux tests ou sources de données) et assure la reproductibilité de l'analyse sensorielle.