Cahier des Charges – Projet Rust-Database (RDB)

Objectif général

Développer en Rust un moteur de stockage Key-Value persistant, log-structuré et concurrent, sans utiliser le mot-clé `unsafe`, sans aide extérieure (IA) et sans dépendances de bases de données existantes.

Ce projet a pour objectifs pédagogiques :

- la compréhension du modèle mémoire de Rust (ownership, borrowing, lifetimes)
- la manipulation bas niveau des fichiers et de l'I/O disque
- la mise en œuvre de structures de données systèmes (index, log, compaction)
- l'implémentation de la concurrence sûre via le compilateur Rust
- l'écriture d'un code idiomatique, robuste et documenté

Contraintes techniques

- Langage : Rust (version stable)
- Bibliothèques autorisées : standard library uniquement (`std`)
- Interdictions :
    - `unsafe`
    - `.unwrap()` et `.expect()` (hors tests)
    - moteurs de bases de données existants
- Tests : `cargo test`
- Plateformes cibles : Linux et macOS

Planning du projet

Semaine 1 – Le moteur fondamental (mono-thread)

Objectif : produire un moteur de stockage persistant fonctionnel dans un contexte mono-thread.

Jour 1 – Immersion Rust et initialisation

Objectif pédagogique
Maîtriser les bases du langage Rust et comprendre le modèle d'ownership.

Tâches :

- Lecture des chapitres 1 à 4 du Rust Book
- Installation et pratique avec rustlings (ownership, borrowing)
- Initialisation du projet :
  `cargo init rust-database`

Livrable :

- Projet compilable sans warnings bloquants
- Présence d'une structure de configuration (ex. `DatabaseConfig`)
- Organisation minimale du projet (`main.rs`, `lib.rs`)

Critères de validation :

- Code clair et commenté
- Aucune panique possible au lancement

Jour 2 – Format binaire et I/O disque

Objectif pédagogique
Définir comment les données sont écrites physiquement sur disque.

Tâches :

- Définition d'un format binaire explicite pour une entrée :
  - taille de la clé
  - taille de la valeur
  - données de la clé
  - données de la valeur
  - checksum simple
- Sérialisation manuelle en `Vec<u8>`
- Écriture séquentielle append-only dans un fichier
- Gestion complète des erreurs avec `Result`

Livrable :

- Fichier binaire contenant plusieurs entrées

- Fonction d'écriture `append_entry(...)`

Critères de validation :

- Aucun `.unwrap()`
- Erreurs I/O correctement propagées

Jour 3 – Index en mémoire

Objectif pédagogique
Permettre un accès direct aux données sans parcourir tout le fichier.

Tâches :

- Implémentation d'un index mémoire :
  `HashMap<String, (offset, size)>`
- Utilisation de Seek pour accéder à un offset précis
- Séparation claire entre logique disque et logique mémoire

Livrable :

- Fonction `get(key)` fonctionnelle
- Lecture directe sans scan complet

Critères de validation :

- Accès correct après plusieurs écritures
- Index cohérent avec le fichier

Jour 4 – CRUD et tombstones

Objectif pédagogique
Finaliser les opérations logiques de la base.

Tâches :

- Implémentation de :
  - `set(key, value)`
  - `get(key)`
  - `delete(key)`
- Suppression logique via un marqueur tombstone
- Gestion des clés inexistantes avec `Option`

Livrable :

- Moteur CRUD complet en mono-thread

Critères de validation :

- Les clés supprimées ne sont plus accessibles
- Aucune suppression physique dans le fichier

Jour 5 – Interface CLI et robustesse

Objectif pédagogique
Rendre le moteur utilisable et résilient aux erreurs.

Tâches :

- Création d'une interface CLI en mode REPL :
  - SET key value
  - GET key
  - DELETE key
  - EXIT
- Gestion des cas d'erreur :
  - disque plein
  - fichier corrompu
  - commandes invalides

Livrable :

- Application console fonctionnelle
- Messages d'erreur explicites

Critères de validation :

- Aucun crash possible par entrée utilisateur
- Fermeture propre des fichiers

Semaine 2 – Systèmes avancés

Objectif : améliorer les performances, assurer la concurrence et la fiabilité à long terme.

Jour 6 – Concurrence : fondations

Objectif pédagogique
Comprendre le partage sûr de données en Rust.

Tâches :

- Utilisation de `Arc`, `Mutex` et `RwLock`
- Partage contrôlé de l'index et du fichier

Livrable :

- Code compilable avec ressources partagées

Critères de validation :

- Absence de `unsafe`
- Garanties de sûreté imposées par le compilateur

Jour 7 – Multi-threading réel

Objectif pédagogique
Gérer plusieurs accès concurrents.

Tâches :

- Simulation de plusieurs clients via `std::thread`
- Implémentation du pattern Lecteur/Rédacteur
- Écriture exclusive et lectures concurrentes

Livrable :

- Base de données fonctionnelle en environnement multi-thread

Critères de validation :

- Absence de data races
- Aucun deadlock

Jour 8 – Compaction du stockage

Objectif pédagogique
Empêcher la croissance infinie du fichier de log.

Tâches :

- Parcours du fichier de log
- Création d'un nouveau fichier ne contenant que les entrées valides
- Remplacement atomique de l'ancien fichier

Livrable :

- Commande `compact`

Critères de validation :

- Taille du fichier réduite
- Aucune perte de données

Jour 9 – Recovery et fiabilité

Objectif pédagogique
Garantir la persistance après redémarrage.

Tâches :

- Reconstruction automatique de l'index au démarrage
- Vérification des checksums
- Gestion des entrées corrompues

Livrable :

- Base de données persistante après redémarrage

Critères de validation :

- Index reconstruit sans intervention
- Données intactes

Jour 10 – Refactoring et finalisation

Objectif pédagogique
Produire un code idiomatique et professionnel.

Tâches :

- Refactorisation avec :
  - iterators
  - traits
  - modules clairs
- Documentation avec `cargo doc`
- Nettoyage final du code

Livrable :

- Projet final propre, documenté et maintenable

Critères de validation :

- Code lisible et structuré
- Documentation exploitable

Ressources autorisées

- Rust Book : https://doc.rust-lang.org/book/
- Documentation standard : https://doc.rust-lang.org/std/
- Rust by Example : https://doc.rust-lang.org/rust-by-example/