Spécification : Base Historique Géolocalisée – MASTER

Ce document centralise toutes les spécifications fonctionnelles, techniques et stratégiques discutées dans le cadre du projet de base historique géolocalisée. Il s'agit d'un fichier maître cumulant toutes les synthèses validées, étapes de traitement, choix d'architecture et critères d'évaluation utilisés pour créer et exploiter une base robuste liée à des personnages historiques, lieux, et événements majeurs.

# 1. Personnages et thèmes traités

Les sujets actuellement ciblés sont :

- Roland  
- Excalibur  
- Durandal (l'épée)  
- Jean Picard  
- Claude Chappe  
- Guillaume le Conquérant  
- Jeanne d'Arc  
- Jules César  
- Guerre de Cent Ans  
- Aliénor d'Aquitaine  
- Richard Cœur de Lion  
- Philippe Auguste  
- Napoléon  
- Charles VII

# 2. Objectifs de volumétrie

|  |  |
| --- | --- |
| Type de donnée | Objectif |
| Fichiers bruts (pages liées au personnage) | 100 000 lignes réparties sur ~15 fichiers |
| Pages géolocalisées finales | 20 000 lignes |
| Filtres P31 et sous-classes | ~5 000 catégories uniques |
| Filtrage géographique moyen | ~10 éléments dans un rayon de 20 km |

# Pipeline de traitement – Étapes synthétiques

Chaque entrée passe par les étapes suivantes, réparties en modules (ou passes) pouvant être parallélisés par batch de 100 éléments.

|  |  |  |
| --- | --- | --- |
| Étape | Description | API / Source |
| 1 | Extraction des titres de pages liés à un sujet + QID | API MediaWiki (`categorymembers`, `prop=pageprops`) |
| 2 | Enrichissement avec résumé, description, lien Wikipédia | API REST Wikipedia `/page/summary/{title}` |
| 3 | Récupération des types (P31) et calcul de la notoriété | SPARQL Wikidata |
| 4 | Conversion des coordonnées GPS en Lambert 93 | pyproj |
| 5 | Récupération de l’arborescence des P31 via P279 | SPARQL Wikidata |
| 6 | Insertion dans la base de données SQLite | SQLite |
| 7 | Filtrage avancé (pertinence, rayon, types) | Filtrage local |

# Architecture modulaire et parallèle

Le traitement est optimisé pour tourner de nuit, en tâche de fond, avec :  
- Un découpage en batchs de 100 lignes.  
- Des modules qui s'enchaînent via une file d'attente.  
- Une gestion de backlog pour les erreurs ou timeouts.  
- Des logs de progression (une ligne toutes les minutes).  
- Une structure modulaire permettant reprise sur plantage.

# Structure des fichiers JSON

Chaque ligne du fichier contient :

- titre : Titre Wikipédia  
- qid : QID Wikidata  
- lat / lon : Coordonnées GPS  
- x\_l93 / y\_l93 : Coordonnées Lambert 93  
- url : Lien Wikidata  
- wikipedia : Lien Wikipédia  
- description : Description courte  
- resume : Résumé court  
- types : Liste des P31 associés  
- parents : Liste des parents via P279  
- nb\_langues : Nombre d’interwikis  
- note : Score de notoriété (1–10)  
- keep : Booléen de filtrage logique (calculé)

# Architecture modulaire en pipeline (par répertoires)

|  |  |  |
| --- | --- | --- |
| Module | Répertoire d’entrée | Répertoire de sortie |
| 1. Extraction titres (API MediaWiki) | input\_pages/ | qid\_pages/ |
| 2. Ajout QID et coordonnées | qid\_pages/ | enriched\_pages/ |
| 3. Infos Wikipedia (résumé + description) | enriched\_pages/ | wiki\_infos/ |
| 4. Enrichissement SPARQL (types, langues, notoriété) | wiki\_infos/ | semantic\_infos/ |
| 5. Arborescence des types (P31 → P279) | semantic\_infos/ | typed\_infos/ |
| 6. Insertion en base SQLite | typed\_infos/ | db/ |

# Cycle complet de traitement pour un fichier (par étape/module)

|  |  |  |
| --- | --- | --- |
| Étape | Action | Détail |
| 1 | Détection | Scan du répertoire d’entrée pour détecter de nouveaux fichiers. |
| 2 | Verrouillage / Sélection | Renommage ou déplacement temporaire pour éviter les conflits. |
| 3 | Chargement | Lecture du fichier (JSON ou CSV). |
| 4 | Traitement | Application de la logique métier (enrichissement, transformation, etc.). |
| 5 | Écriture | Sauvegarde du résultat dans le répertoire de sortie. |
| 6 | Archivage / Suppression | Déplacement du fichier original dans un répertoire archive/done. |
| 7 | Log | Impression de l’état d’avancement dans un fichier log ou la console. |

# Tableau de correspondance : Étapes du pipeline vs Reader / Process / Writer

| **Étape** | **Reader** | **Processor** | **Writer** |
| --- | --- | --- | --- |
| **1. Extraction des titres MediaWiki (par thème)** | ParamReader | MediaWikiTitleFetcher | CSVWriter |
| **2. Résumés + description Wikipedia (REST)** | CSVReader | WikipediaRESTFetcher | JSONWriter |
| **3. Données Wikidata (QID, coord, URL, P31...)** | JSONReader | WikidataSPARQLFetcher | JSONWriter |
| **4. Arborescence P31 / P279** | JSONReader | P31TreeBuilder | JSONWriter |
| **5. Notoriété + Calcul score (langues interwiki)** | JSONReader | NotorietyScorer | JSONWriter |
| **6. Calcul Lambert93 (x/y)** | JSONReader | LambertProjector | JSONWriter |
| **7. Insertion base SQLite finale** | JSONReader | IdentityProcessor | SQLiteWriter |
| **8. Enrichissement post-filtrage (pertinence locale)** | JSONReader | LocalRelevanceScorer | JSONWriter ou SQLiteWriter |

🧩 1. Classe Reader

| **Classe** | **Rôle** | **Fonctionnalités spécifiques** |
| --- | --- | --- |
| BaseReader | Interface commune | .scanFichiers(), .loadLignes(), .getBatch(n) |
| CSVReader | Lecture fichiers CSV | Parsing CSV ligne à ligne, batch de dictionnaires simples |
| JSONReader | Lecture fichiers JSON | Chargement complet, itération ligne à ligne ou clé par clé |
| ParamReader | Lecture à partir de paramètres | Génère dynamiquement une liste de lignes à traiter |

⚙️ 2. Classe Processor

| **Classe** | **Rôle** | **Fonctionnalités spécifiques** |
| --- | --- | --- |
| BaseProcessor | Interface commune | .processLigne(ligne) ou .processBatch(lignes) |
| MediaWikiTitleFetcher | Appels à l’API MediaWiki pour récupérer les titres | Filtrage par sujet / thème, pagination |
| WikipediaRESTFetcher | Appels à l’API REST Wikipedia | Résumé, description, liens |
| WikidataSPARQLFetcher | SPARQL (QID, coord., URL, P31...) | Requêtes par QID, extractions coordonnées et types |
| P31TreeBuilder | Récupération hiérarchie P31/P279 | Récursivité, cache mémoire, création structure arborescente |
| NotorietyScorer | Nombre de langues et score de notoriété | SPARQL schema:isPartOf, calcul score 1–10 |
| LambertProjector | Conversion GPS → Lambert 93 | Utilise pyproj, ajoute x\_l93 / y\_l93 |
| IdentityProcessor | Pas de traitement, simple transit | Pass-through |
| LocalRelevanceScorer | Calcul de pertinence locale | Analyse de contenu textuel, scoring vs personnage central |

🧾 3. Classe Writer

| **Classe** | **Rôle** | **Fonctionnalités spécifiques** |
| --- | --- | --- |
| BaseWriter | Interface commune | .writeLigne() ou .writeBatch() |
| CSVWriter | Écriture fichiers CSV | Export ligne par ligne, colonnes plates |
| JSONWriter | Écriture fichiers JSON | Sauvegarde par ligne (JSONL) ou liste complète |
| SQLiteWriter | Insertion base SQLite | Mapping vers table, commit en batch |

# Architecture des modules de traitement en pipeline

| **Module** | **Rôle principal** | **Fonctionnalités clés** | **Responsabilités spécifiques** |
| --- | --- | --- | --- |
| BaseReader | Lire les données d'entrée en batch (100 lignes typiquement) | - Scan d’un répertoire d’entrée- Chargement de fichiers CSV ou JSON- Préparation des objets bruts | - Lire les fichiers (1 à la fois)- Extraire les lignes en batch- Marquer le fichier comme “en cours” pour éviter duplication- Transmettre les données au BatchProcessor |
| BatchProcessor | Traiter les objets ligne par ligne avec résilience | - Traitement unitaire avec retry- Gestion des timeouts- Suivi du nombre d’échecs- File d'attente retry\_queue | - Enrichir chaque ligne via REST ou SPARQL- Accumuler les résultats valides- Relancer les échecs jusqu’à 3 fois- Déplacer vers discarded après 3 échecs |
| BaseWriter | Sauvegarder les résultats valides et tracer les erreurs | - Écriture des fichiers .json ou .sqlite- Journalisation des erreurs- Atomicité de l’écriture | - Sauver les lignes traitées avec succès- Sauver les lignes rejetées après 3 essais dans discarded- Nettoyer ou déplacer les fichiers sources traités |

# 🔄 Flux de traitement d’un batch

| **Étape** | **Action** | **Responsable** |
| --- | --- | --- |
| 1 | Le fichier input/nom\_batch.csv est détecté | BaseReader |
| 2 | Les 100 lignes sont lues et converties en objets | BaseReader |
| 3 | Chaque ligne est enrichie (API REST / SPARQL) | BatchProcessor |
| 4 | En cas de succès → ligne envoyée au writer | BatchProcessor |
| 5 | En cas d’échec → stockée dans retry\_queue avec compteur d’essai | BatchProcessor |
| 6 | Lorsque retry\_queue atteint 100 ou en fin de traitement → retraits avec compteur incrémenté | BatchProcessor |
| 7 | Si compteur > 3 → ligne déplacée dans discarded/ | BatchProcessor |
| 8 | Le batch valide est écrit dans output/ (JSON ou SQLite) | BaseWriter |
| 9 | Le fichier source est renommé ou archivé après succès | BaseWriter |

📁 **Tableau de synthèse — Gestion des fichiers par process**

| **Élément** | **Description** | **Exemple / Détail** |
| --- | --- | --- |
| **Répertoire par process** | Un répertoire dédié par étape de traitement (ex: step1\_fetch\_titles/) | step3\_enrich\_wikidata/, step5\_insert\_sqlite/, etc. |
| **Identifiant de run** (run\_id) | Identifiant unique par batch lancé, utilisé comme préfixe de tous les fichiers | JDarc06, Napoleon02, etc. |
| **Nom de fichier** | Format standardisé pour garantir l’ordre et la traçabilité | JDarc06\_step3\_001.json |
| **Index séquentiel** | Compteur croissant dans le nom de fichier pour identifier l’ordre et détecter les trous | 001, 002, etc. |
| **Sous-répertoire done/** | Chaque process déplace les fichiers traités dans done/ pour archivage | step2\_fetch\_summaries/done/JDarc06\_step2\_003.json |
| **Fichier résultat** | Une fois traité, le fichier est déplacé dans le répertoire du process suivant | step3\_enrich\_wikidata/JDarc06\_step2\_003.json |
| **Pas de suppression automatique** | Aucun fichier source n’est supprimé automatiquement : uniquement des move (déplacement) | Sécurité assurée en cas de redémarrage, audit, ou vérification |
| **Log par process** | Fichier de log dédié à chaque process avec rotation si besoin | logs/step3\_enrich\_wikidata.log |
| **Contrôle d’unicité run\_id** | Vérification qu’un même run\_id n’est pas traité plusieurs fois par erreur | Ex: JDarc06\_step3\_014.json déjà présent → erreur ou alerte |
| **Pas de timestamp dans le nom** | Les timestamps ne sont **pas** utilisés dans le nom de fichier (trop lourd, peu pertinent) | Préférence pour l’index + run\_id |
| **Eventuel timestamp interne** | Peut être ajouté dans le contenu JSON si besoin de traçabilité temporelle | Clé "timestamp": "2025-05-27T15:42:00Z" dans le contenu du fichier |

**Stratégie d’Extraction et de Filtrage**

**Étape 1 – Extraction large**

L’objectif de cette première étape est de **ratisser aussi large que possible** afin de ne rater aucun article potentiellement pertinent.  
Cela implique l'utilisation de requêtes plein texte via l’API Wikipedia (list=search), en recherchant les occurrences du mot-clé dans le **contenu complet des articles**, et pas seulement dans les titres ou les catégories.  
Cette approche inclut volontairement **un niveau élevé de bruit** (par exemple : écoles, clubs, rues, établissements portant le nom du personnage ciblé).

**Étapes 2 à 6 – Enrichissement progressif**

Les étapes suivantes ajoutent les **métadonnées structurantes** nécessaires à la sélection intelligente :

* QID Wikidata
* Coordonnées géographiques (latitude/longitude + Lambert 93)
* Typage (P31) et hiérarchie de parenté (P279)
* Indicateurs de notoriété (langues, longueur de résumé, etc.)

Grâce à ces données, un **filtrage fin** peut ensuite être appliqué en aval, selon le contexte :

* **Filtrage géographique** (ex : uniquement les objets dans un rayon de 100 km)
* **Filtrage sémantique** (ex : garder les événements historiques, lieux géographiques, etc.)

**Justification**

Cette séparation entre **extraction large** et **filtrage intelligent** garantit :

* La **non-perte d’information** dans les premières étapes
* Une **modularité** dans la logique de traitement (possibilité de réajuster les filtres sans relancer l’extraction)
* Une **performance maîtrisée** : le volume initial est stocké en JSONL, filtré ensuite avant insertion SQLite

Ce découpage permet également de rejouer l'étape de sélection ou d'affinage ultérieurement avec de nouveaux critères (notoriété, typologie, zone géographique) sans avoir à repartir de zéro.

**Stratégie d’Extraction et de Filtrage**

| **Étape** | **Dossier source** | **Dossier de sortie** |
| --- | --- | --- |
| 1 | *(paramétrique / CLI)* | step1\_backlinks/ |
| 2 | step1\_backlinks/ | step2\_coords/ |
| 3 | step2\_coords/ | step3\_summary/ |
| 4 | step3\_summary/ | step4\_semantics/ |
| 5 | step4\_semantics/ | step5\_types/ |
| 6 | step5\_types/ | db/ |
| 7 | db/ | *(résultat filtré ou SQLite)* |