**Plan d'action – Enrichissement des P31 et arborescence P279 via API REST**

**Objectif :** Remplacer totalement SPARQL par des appels REST batchés pour enrichir les entités P31 et construire dynamiquement leur arborescence P279, avec stockage dans une base SQLite.

**✅ Étape 1 : Identification des P31 manquants**

* Scanner tous les fichiers d’entrées (ex : step4\_\*.json)
* Extraire la liste unique des P31 non présents dans la base SQLite
* Batcher cette liste par paquets de 50 QID

**✅ Étape 2 : Récupération REST des P279**

* Pour chaque QID P31, faire un appel REST groupé (50 max)
* Lire les propriétés P279 ("subclass of") pour construire la chaîne ascendante
* Détecter récursivement les parents jusqu'à un QID déjà présent en base

**✅ Étape 3 : Stockage en base SQLite**

* Table P31Tree avec colonnes :
  + qid: QID enfant (ex : Q43229)
  + parent: QID parent direct
  + profondeur: 0 pour le plus haut, +1 par niveau descendant
* Index sur qid et parent

**✅ Étape 4 : Mise à jour de la base**

* Charger les QID inconnus et ajouter leur arborescence
* Toujours tester la présence du parent avant d’ajouter une nouvelle chaîne
* Éviter les cycles

**✅ Étape 5 : Mode fichier temporaire (facultatif)**

* Enregistrer les résultats de batch dans un p31Tree\_temp.json si besoin
* Utiliser ce fichier pour intégration à chaud ou débogage

**✅ Outils et bonnes pratiques**

* Rester 100% REST pour les enrichissements
* Interrogation SPARQL seulement pour exploration ponctuelle
* Toujours utiliser les fonctions requeteWikiMedia() et batching centralisées

**Démarrage demain :**

* Implémenter un nouveau BatchProcessingP31Tree (inspiré de l'étape 3/4)
* Construire l’appel batch vers /entities/ et parser les claims.P279
* Intégrer la chaîne d’appel dans un wikiDataLoader\_Etape5.py
* Ajouter l’étape 5 dans le main.py avec listener automatique

Prêt à redémarrer demain matin avec performance et cohérence !

**Titre : Algorithme d'enrichissement de l'arborescence P31 / P279 via API REST**

**Objectif** : Construire et enrichir l'arborescence des types Wikidata (relations P31 / P279) en partant des types P31 de nouvelles entrées historiques, récupérés par lots via l'API REST, tout en évitant les redondances et en garantissant la cohérence hiérarchique.

**1. Prérequis**

* Une base SQLite existante contenant la table ArborescenceP279(qid, parent\_qid, profondeur, visible)
* Des fichiers JSON contenant des objets EntreeHistorique avec le champ p31
* Une méthode REST batch pour /wbgetentities?ids=...&props=claims

**2. Déroulement de l'algorithme**

**Étape 1 : Chargement des données existantes**

*  Lire tous les fichiers JSON d’un répertoire donné (ex: step4\_semantics/)
*  Extraire les p31 présents dans chaque EntreeHistorique
*  Conserver l’unicité via un set(p31)
* Charger en mémoire l'ensemble des qid déjà présents dans la table ArborescenceP279
  + Utiliser un set pour accélérer la recherche

**Étape 2 : Identification des P31 inconnus**

* Lire tous les fichiers d'entrée et extraire les p31 uniques
* Filtrer ceux absents de la base existante
* Ajouter ces QID dans une liste à traiter

**Étape 3 : Appels REST en batch**

* Par paquet de 20 QID (maximum supporté par l'API), appeler /wbgetentities
* Pour chaque entité retournée, extraire ses P279 (parents directs)
* Stocker les noeuds dans une structure temporaire en mémoire :

class NoeudArbo:

qid: str

parent: Optional[str]

enfants: List[str]

profondeur: Optional[int] = None

visible: bool = True

* Si le parent est connu en base, relier le noeud
* Sinon, ajouter le parent à la liste des QID à traiter

**Étape 4 : Construction progressive de l'arborescence**

* Répéter l'étape 3 tant qu'il reste des parents inconnus
* Une fois tous les parents connus ou atteints, calculer les profondeurs :
  + Profondeur 0 = racines connues de la base
  + Chaque enfant a profondeur = parent + 1

**Étape 5 : Détection des racines temporaires**

* Un noeud est une racine temporaire si son parent n'existe pas dans le graphe temporaire **et** est connu en base
* Ces racines servent de point de départ pour la propagation de la profondeur

**Étape 6 : Insertion en base**

* Pour chaque noeud de l'arbre temporaire, dans l'ordre des profondeurs croissantes :
  + Vérifier qu'il n'existe pas déjà en base (via qid)
  + INSERT INTO ArborescenceP279 (qid, parent\_qid, profondeur, visible)

**3. Précautions et cas particuliers**

* ❌ Éviter les cycles : garder un ensemble visites\_en\_cours pour détecter une boucle
* ⚡ Optimiser les performances en mémoire : ne pas réinterroger des QID déjà analysés
* ✅ Ne jamais supprimer les relations existantes en base
* ❄️ Ne traiter que les P31 utiles pour les nouvelles entrées

**4. Améliorations futures potentielles**

* Ajout d'une table de cache local pour les labels P31 / P279
* Visualisation graphique de l'arborescence
* Ajout de marqueurs thématiques (ex: historique, militaire, religieux) via d'autres propriétés

**Conclusion** Cet algorithme garantit une construction progressive, cohérente et performante de l'arborescence typologique des entrées historiques. Il permet un enrichissement massif en minimisant les appels répétitifs à l'API et en assurant une traçabilité par profondeur et visibilité.