Récupération et intégration de lots de données depuis le GBIF

Fonctionnement global

1. Configurer et déclencher un export ciblé via l'API du GBIF
2. Récupérer le fichier généré sur le serveur
3. Préparer les données et les stocker dans la base de données
4. Préparer les métadonnées associées
5. Intégrer ou actualiser les données dans la synthèse

Scripts

Configurer l'export GBIF

La récupération d'un grand volume de données pré-filtrées selon différentes variables est possible en configurant un export via l'API download du GBIF (ici en json). Le fichier de configuration est transmis à l'API du GBIF, qui met ensuite à disposition un export (fichiers) dans l'espace utilisateur du demandeur, correspondant à la requête configurée. Il est donc nécessaire d'avoir un compte GBIF actif.

Dans le fonctionnement du pôle invertébrés, les données filtrées concernent les données : * disposant de coordonnées géographiques * sans problème géospatial (géométries invalides etc) * situées en Auvergne-Rhône-Alpes * d'un ou certains jeux de données ciblés à l'avance * des groupes (phylum) les mieux connus de la faune invertébrée

Pour connaitre les "PHYLUM_KEY" à filtrer, les correspondances avec taxref sont disponibles par défaut dans les bases de données GeoNature dans taxonomie.taxref_liens. Par exemple pour les invertébrés :

```
SELECT DISTINCT t.cd_nom, tl.ct_sp_id
FROM taxonomie.taxref t
JOIN taxonomie.taxref_liens tl ON tl.cd_nom = t.cd_nom
WHERE tl.ct_name ='GBIF'
AND t.id_rang = 'PH'
AND t.regne='Animalia'
AND t.phylum!='Chordata';
```

La configuration de l'export doit être stockée dans un fichier json. À titre d'exemple pour récupérer les Annélides, Mollusques et Arthropodes du jeu de données iNaturalist, le fichier query.json comporte la requête suivante :

```
{
 "creator": "USER",
 "notificationAddresses": [
```

```
 "EMAIL"
],
"sendNotification": true,
"format": "DWCA",
"predicate": {
"type": "and",
"predicates": [
 {
  "type": "equals",
  "key": "HAS_COORDINATE",
  "value": "True"
 },
 {
  "type": "equals",
  "key": "HAS_GEOSPATIAL_ISSUE",
  "value": "False"
 },
 {
  "type": "equals",
  "key": "GADM_GID",
  "value": "FRA.1_1"
 },
 {
  "type": "in",
  "key": "DATASET_KEY",
  "values": [ "50c9509d-22c7-4a22-a47d-8c48425ef4a7" ]
 },
 {
  "type": "in",
  "key": "PHYLUM_KEY",
  "values": [ "42", "52", "54" ]
 }
]
}
}
```

Serveur : Récupérer, préparer et stocker les données en base

Une fois le fichier query.json configuré, la requête est transmise à l'API à l'aide de la commande suivante (installation du package jq pour une meilleure lisibilité des réponses) :

```
sudo apt install jq
curl –silent --user USER_GBIF:PASS_GBIF --header "Content-Type:
application/json" --data @query.json
https://api.gbif.org/v1/occurrence/download/request | tail -n 1 | tr -d '\r'
```

Cette commande retournera la clé de l'export en question, sous une forme telle que : 0015000-260108000000665

On peut ensuite suivre l'état de la requête (elle peut prendre quelques dizaines de minutes) avec la commande suivante :

```
curl -Ss https://api.gbif.org/v1/occurrence/download/<CLE> | jq -r '.status'
```

Une fois le statut "SUCCEDED", se placer dans le répertoire /tmp et procéder au téléchargement des données :

```
cd /tmp
curl --location --remote-name
https://api.gbif.org/v1/occurrence/download/request/<CLE>.zip & unzip
<CLE>.zip
```

Une fois les données récupérées, les fichiers seront nettoyés des champs non utilisés pour optimiser les temps de traitement et l'espèce occupé sur le serveur. Deux fichiers en particulier seront exploités : occurrence.txt et multimedia.txt

```
cut -f
1,6,8,18,22,25,27,30,31,32,33,34,35,40,48,63,90,98,99,100,135,149,180,188,19
4,206 /tmp/occurrence.txt > data_gbif_import.txt
cut -f 1,4 /tmp/multimedia.txt > multimedia_import.txt
```

Enfin, deux tables de destination sont créées dans la base de données (au premier usage) :

```
CREATE SCHEMA pinv_gbif;

CREATE TABLE pinv_gbif.tmp_gbif_data (
gbif_id bigint primary key,
modified timestamp,
"references" varchar(255),
basis_of_record varchar(255),
occurence_id varchar(255),
recorded_by varchar(255),
individual_count varchar(255),
sex varchar(255),
life_stage varchar(255),
reproductive_condition varchar(255),
caste varchar(255),
behavior varchar(255),
vitality varchar(255),
occurrence_status varchar(255),
occurrence_remarks text,
event_date timestamp,
verbatim_locality varchar(255),
decimal_latitude numeric,
decimal_longitude numeric,
coordinate_uncertainty_meters varchar(255),
identified_by varchar(255),
scientific_name varchar(255),
dataset_key uuid,
```

```
issue text,
taxonKey int,
verbatim_scientific_name varchar(255)
);

CREATE TABLE pinv_gbif.cor_multimedia (
gbif_id bigint,
media_url text
);
```

Puis alimentées par les données récupérées :

```
sudo su postgres
psql -d <DB_name>
\copy pinv_gbif.tmp_gbif_data
FROM PROGRAM 'tail -n +2 /tmp/data_gbif_import.txt'
WITH (
FORMAT text,
DELIMITER E'\t'
);

\copy pinv_gbif.cor_multimedia
FROM PROGRAM 'tail -n +2 /tmp/multimedia_import.txt'
WITH (
FORMAT text,
DELIMITER E'\t'
);
```

Étape intermédiaire iNaturalist

iNaturalist génère des uuid pour ses observations, exportables depuis la plateforme pour des faibles volumes de données, mais non traités/restitués par le GBIF. iNaturalist invite cependant à télécharger les données depuis le GBIF et non pas leur plateforme en direct pour des questions de performance de leur infrastructure...

Le choix du Pôle invertébrés est donc :

- de récupérer les occurrences de taxons depuis le GBIF pour partager les mêmes scripts avec d'autres jeux de données, et pour ne pas dépasser les capacités de service de iNaturalist
- de récupérer les identifiants de données iNaturalist
- de récupérer uniquement les uuid de ces données sur l'API v2 de iNaturalist (pas optimal, mais moindre mal), afin de ne pas générer de nouveaux uuid à ces données

Les identifiants des données iNaturalist récupérées auprès du GBIF sont isolés :

```
SELECT id_synthese, digital_proof, REPLACE(replace(digital_proof,
'https://www.inaturalist.org/observations/',''),
'http://www.inaturalist.org/observations/','') AS id_inaturalist FROM
gn_synthese.synthese s
```

```
JOIN taxonomie.taxref t ON t.cd_nom=s.cd_nom
WHERE id_dataset=3125;
```

Ces id_inaturalist sont ensuite utilisés dans R, avec le script suivant, pour récupérer les uuids des données dans iNaturalist, et pouvoir les réimporter dans la synthèse ensuite.

fetch_inat_uuid ← function(

```
  df,
  id_col = "id_inaturalist",
  uri_col = "url",
  uuid_col = "uuid",
  batch_size = 50,
  sleep_sec = 1
```

) {

```
library(httr)
library(jsonlite)
library(progress)

# typer les id_inaturalist en integer
df[[id_col]] <- as.integer(trimws(df[[id_col]]))

if (!uuid_col %in% names(df)) {
  df[[uuid_col]] <- NA_character_
}

if (!uri_col %in% names(df)) {
  df[[uri_col]] <- NA_character_
}

ids_to_fetch <- df[[id_col]][is.na(df[[uuid_col]])]

batches <- split(ids_to_fetch, ceiling(seq_along(ids_to_fetch) /
batch_size))

pb <- progress_bar$new(
  format = "  :bar :percent | batch :current/:total | ETA: :eta",
  total = length(batches),
  clear = FALSE,
  width = 60
)

for (batch_ids in batches) {
  res <- try(
    GET(
      "https://api.inaturalist.org/v2/observations",
      query = list(
        id = paste(batch_ids, collapse = ","),
        fields="id,uri,uuid",
```

```
        per_page = length(batch_ids)
      ),
      user_agent("inat-uuid-fetcher/1.0 (contact: donovan.maillard@flavia-
ape.fr)")
    ),
    silent = TRUE
  )
  if (!inherits(res, "response") || status_code(res) != 200) {
    pb$tick()
    Sys.sleep(sleep_sec * 2)
    next
  }
  json <- fromJSON(
    content(res, as = "text", encoding = "UTF-8"),
    flatten = TRUE
  )
  if (length(json$results) > 0) {
    uuids <- setNames(
      json$results$uuid,
      as.integer(json$results$id)
    )
    uris <- setNames(
      json$results$uri,
      as.integer(json$results$id)
    )
    idx <- match(as.integer(names(uuids)), df[[id_col]])
    df[[uuid_col]][idx] <- uuids
    idx <- match(as.integer(names(uris)), df[[id_col]])
    df[[uri_col]][idx] <- uris
  }
  pb$tick()
  Sys.sleep(sleep_sec)
}

return(df)
```

}

inaturalist_uuids ← fetch_inat_uuid(

```
inaturalist_ids,
id_col   = "id_inaturalist",
uuid_col = "uuid",
batch_size = 50,
sleep_sec = 1
```

)

Postgresql : Insérer ou actualiser les données en synthèse

À partir des données stockées dans les tables temporaires, l'objectif est de convertir ces données vers le format SINP, et d'alimenter la synthèse. Une fonction d'UPSERT est créée pour insérer les nouvelles données, et actualiser celles déjà disponibles. Dans les 2 cas, il est nécessaire que les jeux de données récupérées auprès du GBIF aient déjà un équivalent dans les métadonnées de l'instance GeoNature, avec le même uuid que celui du dataset dans la base de données du GBIF.

Il faut ensuite disposer d'une source "GBIF" dans gn_synthese.t_sources :

```
INSERT INTO gn_synthese.t_sources()
VALUES ();
```

Pour permettre l'UPSERT, il est nécessaire d'avoir une contrainte d'unicité entre l'id_source correspondant au GBIF, et la colonne entity_source_pk_value, qui correspondra au gbif_id.

```
ALTER TABLE gn_synthese.synthese ADD CONSTRAINT WHERE id_source=X;
```

Une fois le ou les jeux de données préparés et bien identifiés (uuid), créer la fonction suivante :

```
TODO
```

Puis déclencher l'UPSERT :

```
SELECT pinv_gbif.upsert_gbif_data();
```