Collec-Science : stratégies mises en œuvre dans Postgresql

Éric Quinton – Christine Plumejeaud
eric.quinton@irstea.fr
christine.plumejeaud-perreau@univ-lr.fr

IRSTEA Bordeaux – UR *Écosystèmes aquatiques et changements globaux*
UMR LIENSs – *Université de La Rochelle* – CNRS
Document distribué sous licence CC-BY-SA
https://creativecommons.org/licenses/by-sa/3.0/fr/legalcode

Réunion RBDD – 14 décembre 2017

1. Métadonnées
   - Présentation de Collec-Science
   - gestion des données « métier »
   - modèle entité – attribut – valeur
   - implémentation

2. Recherche « plein texte »
   - Types d'index
   - Vacuum
   - Recherche dans un champ JSON

Collec-Science : une appli pour gérer des échantillons

Type	Caractéristiques	Exemples
Collections patrimoniales	Données ouvertes, partagées, base centralisée, entrée par la taxonomie	Recolnat (muséum), Cyber-carothèque
Analyses de labo en routine	échantillons détruits après analyse	LIMS
Échantillons collectés dans le cadre de projets de recherche	Durée de conservation longue (> 40 ans), échanges avec d'autres labos possible	Collec-Science, Barcode
Matériel d'exp. (terrain, aquariums...)	Gestion de stock	Collec-Science
Matériel de laboratoire	métrologie, suivi de l'entretien, assurance-qualité	Split
Bases documentaires	prêt, recensement, mise à disposition : gestion de bibliothèque	PMB

- suivi des échantillons : usages, échantillons dérivés, métadonnées associées, etc.
- gestion des mouvements (entrée, sortie) : étiquettes avec QRCODE, lecture par douchette, etc.

Intégration des métadonnées

- principes initiaux dans Collec-Science
  - application de gestion d'échantillons *stricto sensu*
  - pas de données « métier »
    - à chacun son job !

Intégration des métadonnées

- principes initiaux dans Collec-Science
  - application de gestion d'échantillons *stricto sensu*
  - pas de données « métier »
    - à chacun son job !
- mais...
  - on a parfois besoin de mieux caractériser les échantillons
    - taxon
    - conditions de collecte (opérateur, parcelle, etc.)
  - ces données sont variables selon les types d'échantillons

Intégration des métadonnées

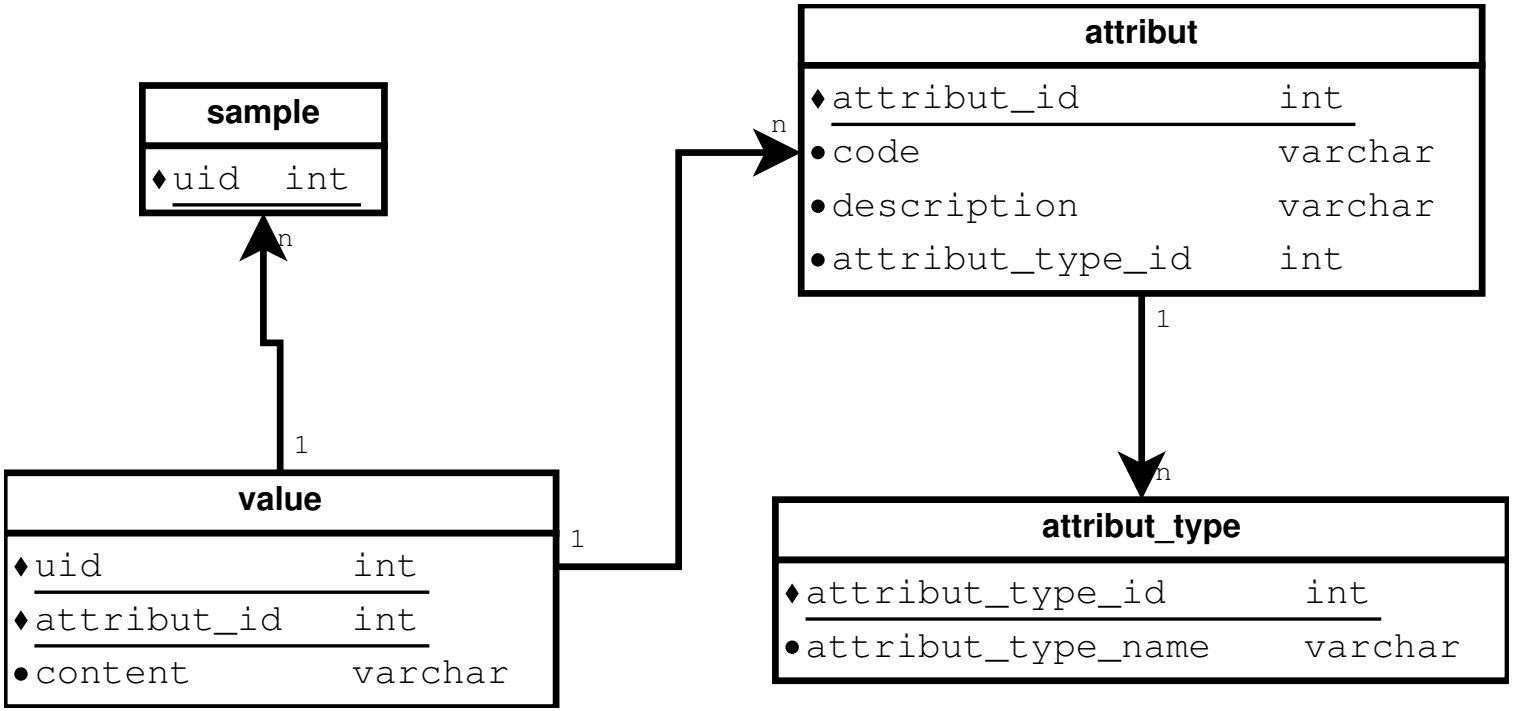
- principes initiaux dans Collec-Science
  - application de gestion d'échantillons *stricto sensu*
  - pas de données « métier »
    - à chacun son job !
- mais...
  - on a parfois besoin de mieux caractériser les échantillons
    - taxon
    - conditions de collecte (opérateur, parcelle, etc.)
  - ces données sont variables selon les types d'échantillons
- décision : rajouter la possibilité de saisir des métadonnées (ou données « métier » associées)

Caractéristiques des métadonnées

- les métadonnées :
  - peuvent varier d'un type d'échantillons à un autre
  - il ne faut pas recourir à des informaticiens pour les définir
  - elles doivent pouvoir être imprimées sur les étiquettes ou figurer dans les QRCodes
- mais :
  - elles doivent rester de structure simple
  - pas question de concurrencer les applications dédiées !

Le modèle entité – attribut – valeur

- extension du modèle entité – relation
  - modèle relationnel classique : tables comprenant des attributs, reliées entre elles
- les colonnes n'ont plus de signification « métier »
  - les valeurs sont associées à un attribut, qui décrit le contenu, voire le type



Inconvénients

- difficulté de typer les données
  - soit tout est stocké en varchar
    - nécessite des fonctions de traduction de type
  - soit on définit en plus des types int, double, timestamp
    - les données sont éparpillées dans plusieurs colonnes
- les requêtes SQL ne sont pas adaptées
  - difficulté de représenter les informations sous la forme d'un tableau
    - requêtes complexes utilisant soit les *array*, soit les *crosstab*
- nomenclature forcément partagée entre tous les utilisateurs de l'application
  - risque non négligeable de conflits entre des secteurs d'activité variés
    - définition différente pour le même attribut...

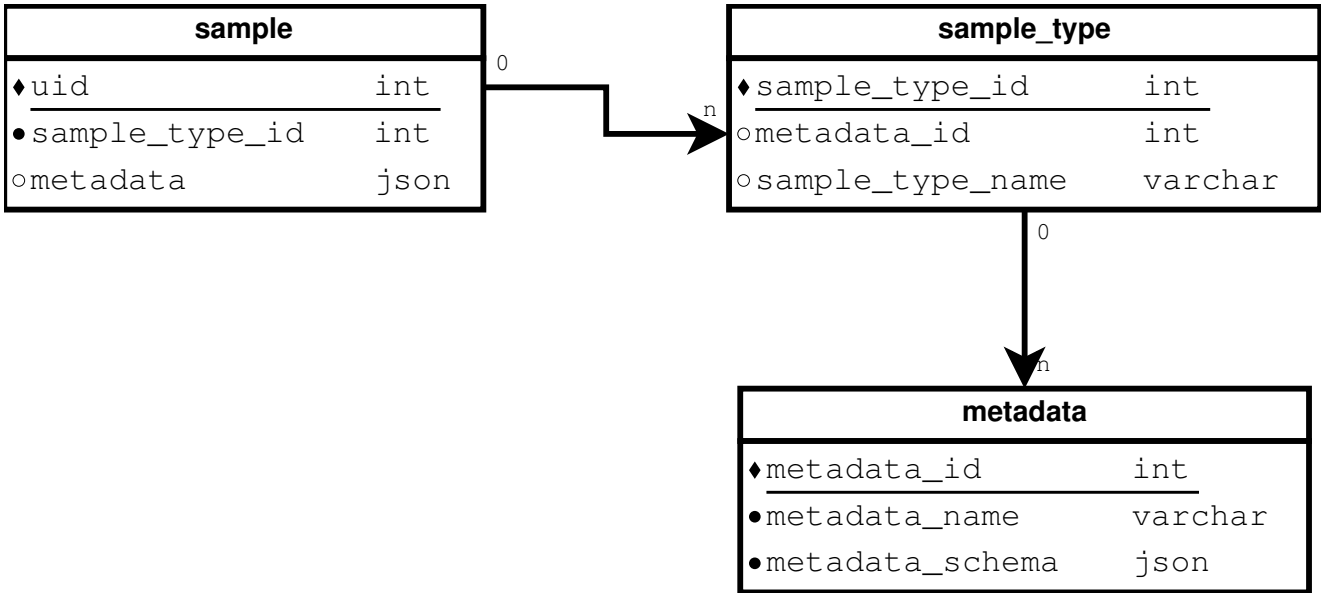
Inconvénients

- difficulté de typer les données
  - soit tout est stocké en varchar
    - nécessite des fonctions de traduction de type
  - soit on définit en plus des types int, double, timestamp
    - les données sont éparpillées dans plusieurs colonnes
- les requêtes SQL ne sont pas adaptées
  - difficulté de représenter les informations sous la forme d'un tableau
    - requêtes complexes utilisant soit les *array*, soit les *crosstab*
- nomenclature forcément partagée entre tous les utilisateurs de l'application
  - risque non négligeable de conflits entre des secteurs d'activité variés
    - définition différente pour le même attribut...
- décision : utiliser un format de stockage structuré
  - format JSON natif avec PostgreSql

Principe retenu pour le stockage des métadonnées

- stockage des métadonnées au format JSON
- PgSql propose 2 types :
  - json : format de type texte, l'info est conservée comme elle est entrée
    - moins performant, redondance d'un attribut possible, etc.
    - avantage : l'ordre de saisie est conservé
  - jsonb : format binaire
    - plus performant, notamment en recherche
    - les attributs sont réordonnés
  - choix retenu : json
    - l'ordre de déclaration des attributs est important et ne doit pas être modifié automatiquement
    - la volumétrie attendue n'est pas si importante qu'elle impose de recourir au jsonb

Principes retenus pour la description des métadonnées



- les types de métadonnées sont rattachés aux types d'échantillons
- *sample.metadata* contient les valeurs « métier »
- *metadata.metadata_schema* contient la description des champs
  - utilisé par *AlpacaJS*

Implémentation

- utilisation de la bibliothèque Javascript AlpacaJS pour gérer les métadonnées
  - génère automatiquement un formulaire de saisie
- la description des métadonnées est stocké dans un champ JSON
  - attributs utilisés pour décrire un champ :
    - *type* (texte, date, liste de sélection, boite à cocher, multiligne)
    - *name* : nom du champ
    - *required* : champ obligatoire ou non
    - *helperChoice* et *helper* : texte d'aide
    - *description* : description complète du champ
    - *measureUnit* : unité de mesure (unité utilisée pour la quantification)
    - *isSearchable** : précise si le champ est utilisé comme critère de recherche ou non
      l'attribut sert à générer un index
    - *choiceList* : liste des valeurs possibles pour les listes de sélection

* *en cours de développement*

Problèmes posés par la recherche « plein texte »

- les échantillons peuvent être recherchés sur plusieurs critères
  - identifiant métier, type, etc.
- la recherche par identifiant utilise une zone de texte
  - recherche plein texte :
    - critère utilisé : lower(identifier) like lower('%value%')
- problème de performance :
  - nécessite de recourir à des index
  - les index de type *btree* ne peuvent pas être utilisés avec *like '%* (pas de signe % possible en début de chaîne)
- le recours à un autre type d'index est nécessaire

Les index GIN et GIST

- prévus pour les recherches « plein texte »
  - peuvent être utilisés sur des vecteurs (*tsvector*)
- extension intégrée à Postgresql
  - create extension pg_trgm ;
- index GIST :
  - index basé sur la comparaison de valeurs de hashage, de longueur fixe
    - index à perte
    - 3 fois plus long en recherche qu'un index GIN

Les index GIN et GIST

- prévus pour les recherches « plein texte »
  - peuvent être utilisés sur des vecteurs (*tsvector*)
- extension intégrée à Postgresql
  - create extension pg_trgm ;
- index GIST :
  - index basé sur la comparaison de valeurs de hashage, de longueur fixe
    - index à perte
    - 3 fois plus long en recherche qu'un index GIN
- index GIN : indexation complète (par groupe de lettres)
  - 3 fois plus rapide que GIST en recherche, 3 fois plus long en création et 10 fois plus long en modification, 3 fois plus volumineux
  - create index object_lower_identifier_idx on object using gin (lower(identifier) gin_trgm_ops) ;

Les index GIN et GIST

- prévus pour les recherches « plein texte »
  - peuvent être utilisés sur des vecteurs (*tsvector*)
- extension intégrée à Postgresql
  - create extension pg_trgm ;
- index GIST :
  - index basé sur la comparaison de valeurs de hashage, de longueur fixe
    - index à perte
    - 3 fois plus long en recherche qu'un index GIN
- index GIN : indexation complète (par groupe de lettres)
  - 3 fois plus rapide que GIST en recherche, 3 fois plus long en création et 10 fois plus long en modification, 3 fois plus volumineux
  - create index object_lower_identifier_idx on object using gin (lower(identifier) gin_trgm_ops) ;
- en résumé :
  - utilisez GIN si la vitesse de recherche est primordiale
  - utilisez GIST si la vitesse d'écriture est primordiale

Comparaison de recherche avec ou sans index

explain select * from object where lower(identifier) like lower ('%medoc%') ;

- sans index :
  Seq Scan on object (cost=0.00..187.19 rows=7 width=47)
  Filter : (lower((identifier) : :text) '%medoc%' : :text)
  - Cost : temps exprimé en nombre de pages-disque parcourues. Le second chiffre (187.19) correspond au nombre total, le premier au temps de préparation (parcours de l'index, par exemple). *width* représente le nombre de rows par page

Comparaison de recherche avec ou sans index

explain select * from object where lower(identifier) like lower ('%medoc%') ;

- sans index :
  Seq Scan on object (cost=0.00..187.19 rows=7 width=47)
  Filter : (lower((identifier) : :text) '%medoc%' : :text)
  - Cost : temps exprimé en nombre de pages-disque parcourues. Le second chiffre (187.19) correspond au nombre total, le premier au temps de préparation (parcours de l'index, par exemple). *width* représente le nombre de rows par page
- avec index :
  Bitmap Heap Scan on object (cost=28.05..46.77 rows=7 width=47)
  Recheck Cond : (lower((identifier) : :text) '%medoc%' : :text)
  -> Bitmap Index Scan on object_lower_identifier_idx
  (cost=0.00..28.05 rows=7 width=0)
  Index Cond : (lower((identifier) : :text) '%medoc%' : :text)
  - la requête est 4 fois plus rapide à s'exécuter

Optimisation de l'analyseur

- les calculs d'optimisation sont basés sur des données théoriques : nombre moyen d'enregistrements par page, répartition, etc.
- ces données ne sont pas mises à jour en temps réel
- lors des opérations d'écriture (suppression, modification) les anciennes valeurs sont conservées, et doivent être purgées
- c'est le programme *vacuum* qui se charge de ce travail

Optimisation de l'analyseur

- les calculs d'optimisation sont basés sur des données théoriques : nombre moyen d'enregistrements par page, répartition, etc.
- ces données ne sont pas mises à jour en temps réel
- lors des opérations d'écriture (suppression, modification) les anciennes valeurs sont conservées, et doivent être purgées
- c'est le programme *vacuum* qui se charge de ce travail
  - processus automatique par défaut : *autovacuum*
  - dans la pratique, la mise à jour des statistiques n'est pas toujours effectuée suffisamment souvent :
    - le planificateur n'optimisera pas correctement les requêtes
    - temps d'exécution fortement accru

Optimisation de l'analyseur

- les calculs d'optimisation sont basés sur des données théoriques : nombre moyen d'enregistrements par page, répartition, etc.
- ces données ne sont pas mises à jour en temps réel
- lors des opérations d'écriture (suppression, modification) les anciennes valeurs sont conservées, et doivent être purgées
- c'est le programme *vacuum* qui se charge de ce travail
  - processus automatique par défaut : *autovacuum*
  - dans la pratique, la mise à jour des statistiques n'est pas toujours effectuée suffisamment souvent :
    - le planificateur n'optimisera pas correctement les requêtes
    - temps d'exécution fortement accru
- conseil : programmez un *vacuum* manuel régulièrement
- juste après la sauvegarde, par exemple :
  vacuumdb -d "collec" –analyze
  - à exécuter avec l'utilisateur *postgres* en ligne de commande ou à programmer dans la *crontab*

Recherche dans le champ JSON

- fonctions disponibles nativement en PostgreSql, dont :
  - where metadata-»'taxon' like '%alosa%'
    - recherche, dans la colonne *metadata*, une valeur dont l'identifiant vaut *taxon* et dont le contenu comprend la chaîne *alosa*
    - peut être surchargé avec une modification de casse (*upper*() ou *lower*()

Recherche dans le champ JSON

- fonctions disponibles nativement en PostgreSql, dont :
  - where metadata-»'taxon' like '%alosa%'
    - recherche, dans la colonne *metadata*, une valeur dont l'identifiant vaut *taxon* et dont le contenu comprend la chaîne *alosa*
    - peut être surchargé avec une modification de casse (*upper*() ou *lower*()
  - nécessite de créer un index de type GIN pour limiter le temps de recherche
    - doit être créé pour chaque clé utilisée pour la recherche, associé à la fonction de casse correspondante
  - exemple :
    create index sample_lower_metadata_taxon_idx on sample using gin (lower(metadata-»'taxon') gin_trgm_ops) ;

Recherche dans le champ JSON

- fonctions disponibles nativement en PostgreSql, dont :
  - where metadata-»'taxon' like '%alosa%'
    - recherche, dans la colonne *metadata*, une valeur dont l'identifiant vaut *taxon* et dont le contenu comprend la chaîne *alosa*
    - peut être surchargé avec une modification de casse (*upper*() ou *lower*()
  - nécessite de créer un index de type GIN pour limiter le temps de recherche
    - doit être créé pour chaque clé utilisée pour la recherche, associé à la fonction de casse correspondante
  - exemple :
    create index sample_lower_metadata_taxon_idx on sample using gin (lower(metadata-»'taxon') gin_trgm_ops) ;
- au niveau de l'application :
  - le positionnement de l'attribut *isSearch* pour un champ de métadonnées entraîne la création automatique de l'index
  - seuls les champs marqués ainsi sont proposés dans l'interface de recherche