Database reverse engineering and assessment for Android applications

INFOM218 - Année académique 2024-2025

Groupe 1

François Bechet

Thibaut Berg

Anaé De Baets

Carine Pochet

Professeur

Prof. Anthony Cleve

Université de Namur

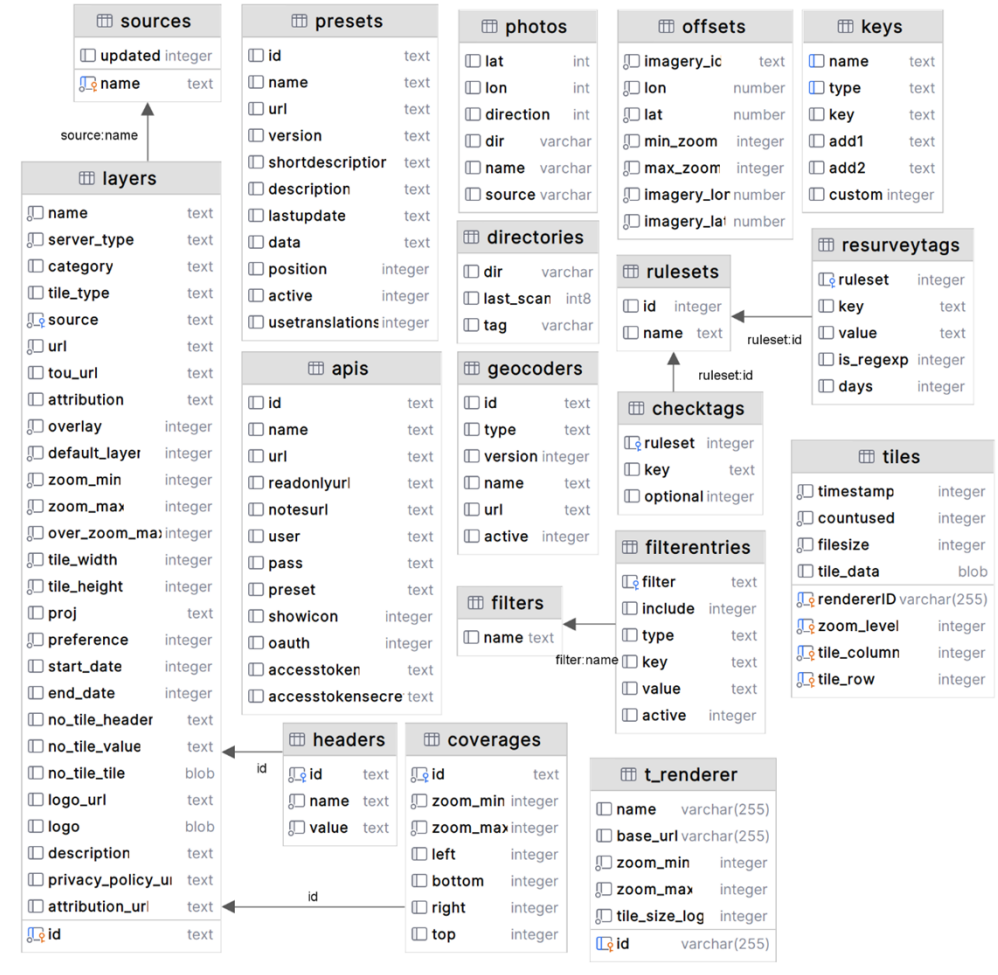
28 décembre 2024

Table des matières

1 Partie 1

1.1 Schéma physique

Pour extraire les données du schéma physique, nous avons utilisé le plugin SQLInspect sur Eclipse afin de récupérer toutes les requêtes SQL du programme. Un script python a ensuite été créé afin de pouvoir trier les différentes requêtes en fonction de mots-clefs. La combinaison des différentes requêtes de création des tables ainsi que des contrainte a mené à ce résultat. On peut y retrouver 6 clefs étrangères qui ne référencent pas toujours des champs uniques. La seule clef primaire composée est située sur la table « tiles ». On retrouve une multitude de champs qui sont nullable. Ces derniers peuvent être utilisés comme référence dans une clef étrangère. Toutes les tables présentent des noms qui sont plutôt explicites. On comprend son utilité et les données qui y sont stockées. Il en va de même pour la majorité des noms d'attributs.



1.2 Analyse supplémentaire pour découvrir des Foreign Keys

1.2.1 Recherche dans le code source

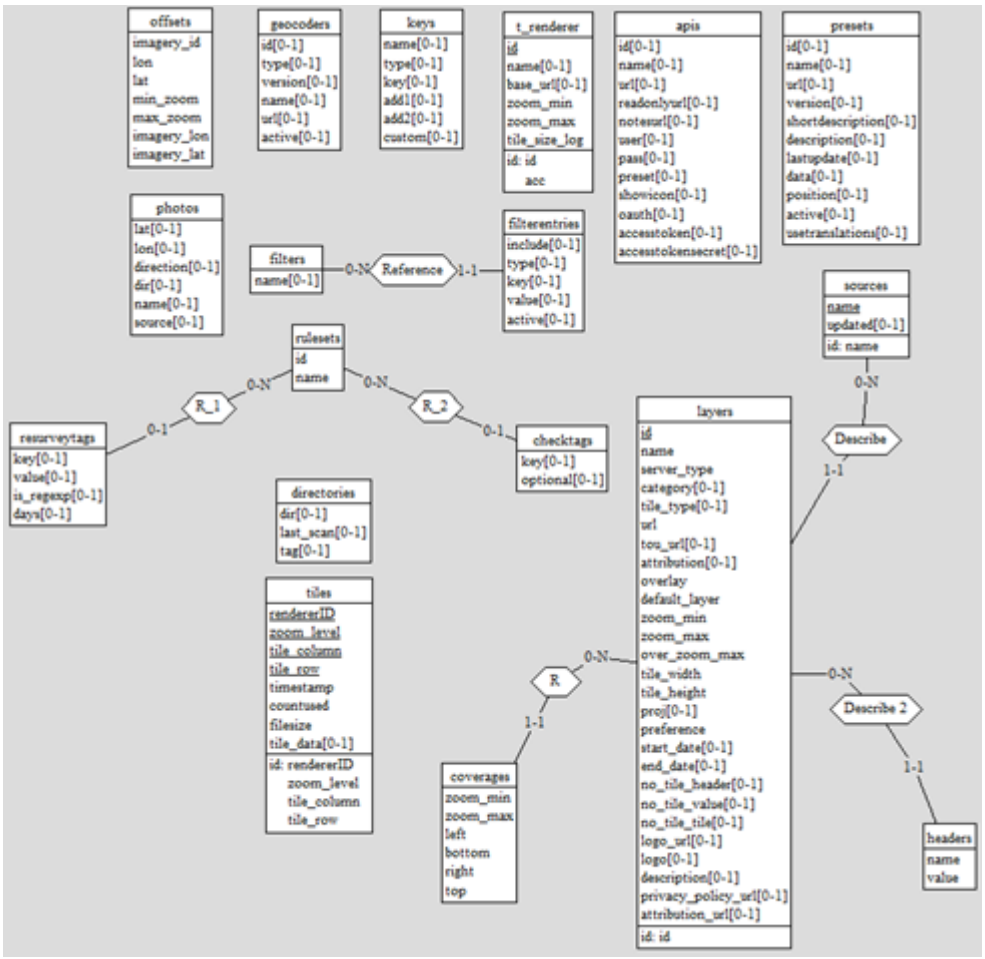
Aucune clef étrangère implicite n'a été trouvée suite à l'analyse des requêtes SELECT avec la regex : « (FROM|from).*,.*(WHERE|where)». Cette dernière permet de sélectionner les requêtes qui réalisent un SELECT sur plusieurs tables. Une seconde analyse des requêtes n'a donné aucun résultat pour le format de jointure SQL2. Une simple recherche du mot clef « JOIN » ne donne aucun résultat, autant dans les requêtes listées par SQLInspect ni dans le code source de l'application.

1.2.2 Recherche de patterns

Les tables « directories » et « photos » ont toutes les 2 un attribut « dir ». En analysant le code du fichier « PhotoIndex.java », et plus précisément de la méthode « private void indexDirectories() », on remarque qu'une query sur la table « photos » utilise la valeur de l'attribut « dir » récupéré à partir d'un select sur la table « directories ». Il y a donc un lien implicite entre ces 2 tables. Toutefois la requête utilise le mot clef « LIKE » qui permet de récupérer les entrés où l'attribut « dir » de la table « photos » contient l'attribut « dire » de la table « directories » suivi ou non d'autres caractères (symbolisé par « % » dans la requête).

1.3 Schéma logique

1.4 Schéma conceptuel


offsets
imagery_id
lon
lat
min_zoom
max_zoom
imagery_lon
imagery_lat
geocoders
id[0-1]
type[0-1]
version[0-1]
name[0-1]
url[0-1]
active[0-1]
keys
name[0-1]
type[0-1]
key[0-1]
add1[0-1]
add2[0-1]
custom[0-1]
t_renderer
id
name[0-1]
base_url[0-1]
zoom_min
zoom_max
tile_size_log
id: id
aoc
apis
id[0-1]
name[0-1]
url[0-1]
readonlyurl[0-1]
noteurl[0-1]
user[0-1]
pass[0-1]
preset[0-1]
showicon[0-1]
oauth[0-1]
accesstoken[0-1]
accesstokensecret[0-1]
presets
id[0-1]
name[0-1]
url[0-1]
version[0-1]
shortdescription[0-1]
description[0-1]
lastupdate[0-1]
data[0-1]
position[0-1]
active[0-1]
usetranslations[0-1]
photos
lat[0-1]
lon[0-1]
direction[0-1]
dir[0-1]
name[0-1]
source[0-1]
filters
name[0-1]
0-N
Reference
1-1
filterentries
include[0-1]
type[0-1]
key[0-1]
value[0-1]
active[0-1]
sources
name
updated[0-1]
id: name
0-N
Describe
1-1
rulesets
id
name
0-N
R_1
0-N
R_2
0-1
0-1
resurveytags
key[0-1]
value[0-1]
is_regexp[0-1]
days[0-1]
checktags
key[0-1]
optional[0-1]
directories
dir[0-1]
last_scan[0-1]
tag[0-1]
tiles
rendererID
zoom_level
tile_column
tile_row
timestamp
countused
filesize
tile_data[0-1]
id: rendererID
zoom_level
tile_column
tile_row
layers
id
name
server_type
category[0-1]
tile_type[0-1]
url
tou_url[0-1]
attribution[0-1]
overlay
default_layer
zoom_min
zoom_max
over_zoom_max
tile_width
tile_height
proj[0-1]
preference
start_date[0-1]
end_date[0-1]
no_tile_header[0-1]
no_tile_value[0-1]
no_tile_tile[0-1]
logo_url[0-1]
logo[0-1]
description[0-1]
privacy_policy_url[0-1]
attribution_url[0-1]
id: id
0-N
R
1-1
coverages
zoom_min
zoom_max
left
bottom
right
top
0-N
Describe 2
1-1
headers
name
value

2 Partie 2

2.1 Statistiques

Type	**# trouvées par SOLInspect**	**# trouvées manuellement**
SELECT	18	24
UPDATE	4	24
DELETE	2	27
INSERT	5	16
CREATE TABLE	18	18
CREATE INDEX	8	8
ALTER TABLE	24	24
Total	**79**	**141**

TABLE 1 – Nombre de requêtes trouvées par type.

Pour les requêtes de la colonne « # trouvées manuellement », un script a été utilisé pour effectuer une analyse statique des requêtes. Il permet de parcourir les différents fichiers Java et, grâce à une regex, les différents appels à SQLite tels que ou encore sont filtrés et enregistrés dans des fichiers séparés.

Pour la complexité de ces dernières, elles sont plutôt simples et courtes. On notera toutefois que les requêtes de création de table peuvent être plus longues, notamment pour la table , mais restent, malgré tout, simples à interpréter.

À propos de la répartition des requêtes dans le projet, elles sont regroupées dans 12 fichiers Java différents. Ces 12 fichiers sont répartis parmi 7 dossiers alors que le projet en compte 58.

Dossier	Fichier

TABLE 2 – Répartition des fichiers contenant des requêtes SQL.

3 Partie 3

3.1 Scénario d'évolution n°1

3.1.1 Scénario

Renommer la colonne de la table .

3.1.2 Modifications

Pour renommer la colonne de la table , les étapes suivantes sont nécessaires. Ces changements concernent le code source et le schéma SQL dans différents fichiers du projet.

1. **Changer la constante qui représente le nom de la colonne**
   - **Fichier concerné** :
   - **Emplacement** : Ligne 82
   - **Ancien code** :

Listing 1 – Ancien code

```
private static final String HEADER_NAME_FIELD = "name";
```

   - **Nouveau code** :

Listing 2 – Nouveau code

```
private static final String HEADER_NAME_FIELD = "newName";
```

2. **Modifier la définition de la table dans la méthode**
   - **Fichier concerné** :
   - **Emplacement** : Ligne 161
   - **Ancien code** :

Listing 3 – Ancien code

```
private void createHeadersTable(SQLiteDatabase db) {
    db.execSQL("CREATE TABLE headers (id TEXT NOT NULL, name TEXT NOT NU
            + " FOREIGN KEY(id) REFERENCES layers(id) ON DELETE CASCADE)
    db.execSQL("CREATE INDEX headers_idx ON headers(id)");
}
```

   - **Nouveau code** :

Listing 4 – Nouveau code

```
private void createHeadersTable(SQLiteDatabase db) {
    db.execSQL("CREATE TABLE headers (id TEXT NOT NULL, newName TEXT NOT
            + " FOREIGN KEY(id) REFERENCES layers(id) ON DELETE CASCADE)
    db.execSQL("CREATE INDEX headers_idx ON headers(id)");
}
```

3. **Mettre à jour la méthode pour refléter le changement**
   - **Fichier concerné** :
   - **Emplacement** : Ligne 716
   - **Ancien code** :

Listing 5 – Ancien code

```
@NonNull
private static Map<String, List<Header>> getHeadersById(SQLi
    Map<String, List<Header>> headersById = new HashMap<>();
    try (Cursor headerCursor = db.rawQuery(
            "SELECT headers.id as id,headers.name as name,va
            new String[] { boolean2intString(overlay) })) {
        if (headerCursor.getCount() >= 1) {
            initHeaderFieldIndices(headerCursor);
            boolean haveEntry = headerCursor.moveToFirst();
            while (haveEntry) {
                String id = headerCursor.getString(headerIdF
                List<Header> headers = headersById.get(id);
                if (headers == null) {
                    headers = new ArrayList<>();
                    headersById.put(id, headers);
                }
                headers.add(getHeaderFromCursor(headerCursor
                haveEntry = headerCursor.moveToNext();
            }
        }
    }
    return headersById;
}
```

— **Nouveau code** :

Listing 6 – Nouveau code

```
@NonNull
private static Map<String, List<Header>> getHeadersById(SQLi
    Map<String, List<Header>> headersById = new HashMap<>();
    try (Cursor headerCursor = db.rawQuery(
            "SELECT headers.id as id, headers.newName as newN
            new String[] { boolean2intString(overlay) })) {
        if (headerCursor.getCount() >= 1) {
            initHeaderFieldIndices(headerCursor);
            boolean haveEntry = headerCursor.moveToFirst();
            while (haveEntry) {
                String id = headerCursor.getString(headerIdF
                List<Header> headers = headersById.get(id);
                if (headers == null) {
                    headers = new ArrayList<>();
                    headersById.put(id, headers);
                }
                headers.add(getHeaderFromCursor(headerCursor
                haveEntry = headerCursor.moveToNext();
            }
        }
    }
    return headersById;
}
```

4 Partie 4

4.1 Schéma de base de données

Mérites

— **Nommage explicite :** Les tables et les colonnes ont des noms descriptifs, donc on sait directement de quoi il s'agit.

— **Utilisation de clés étrangères :** Certaines relations entre les tables utilisent des clés étrangères, assurant l'intégrité référentielle (par exemple, entre les tables et).

— **Segmentation claire des données :** Les tables sont divisées correctement en fonction de leurs fonctionnalités (filtres, photos, headers, etc.).

Inconvénients

1. **Tables et colonnes non utilisées :**
   — La table et sa colonne ne sont pas utilisées.
2. **Colonnes de clés étrangères nullables :**
   — Plusieurs champs de clés étrangères autorisent les valeurs nulles, ce qui peut entraîner des problèmes d'intégrité des données.
3. **Relations implicites :**
   — Certaines relations sont implicites, comme le lien entre et via une requête plutôt qu'une clé étrangère définie.
4. **Données dénormalisées :**
   — Des tables surchargées comme stockent à la fois des métadonnées et des informations géospatiales, réduisant la clarté et la normalisation.
5. **Conception des clés primaires :**
   — La clé primaire pour est composite, ce qui peut compliquer l'indexation et les requêtes.

Recommandations

— Supprimer les tables et colonnes inutilisées, comme , pour réduire la complexité du schéma et améliorer la maintenabilité.

— Définir des clés étrangères explicites pour les relations implicites, comme et , afin d'assurer la cohérence des données.

— Normaliser les données en scindant les tables surchargées (par exemple, diviser en et).

— S'assurer que toutes les clés étrangères font référence à des champs uniques et non nuls pour éviter les problèmes d'intégrité.

— Réévaluer l'utilisation des clés primaires composites et envisager d'introduire des clés substitutives là où c'est pertinent.

4.2 Code de manipulation de la base de données

Avantages

— **Gestion centralisée des requêtes SQL :** Les requêtes sont bien regroupées dans des fichiers Java spécifiques, facilitant la navigation et les mises à jour du code.

— **Requêtes lisibles :** La plupart des requêtes SQL sont simples et directes, facilitant leur compréhension.

Inconvénients

1. **Requêtes codées en dur :**
   — Les requêtes sont souvent codées en dur sous forme de chaînes, ce qui peut poser des défis de maintenance et des risques d'injection SQL.
2. **Utilisation limitée des requêtes paramétrées :**
   — Certaines requêtes utilisent des chaînes concaténées au lieu de la paramétrisation, augmentant les risques pour la sécurité.
3. **Absence de tests unitaires pour les requêtes :**
   — Le manque de tests automatisés pour les instructions SQL peut entraîner des bugs non détectés pendant le développement.
4. **Évolution manuelle du schéma :**
   — Les mises à jour du schéma nécessitent des modifications manuelles des requêtes, augmentant les risques d'erreurs pendant l'évolution du schéma.

Recommandations

— Passer à des requêtes paramétrées ou à un framework ORM (par exemple, pour Android) pour améliorer la sécurité et la maintenabilité.

— Développer une suite de tests complète pour les opérations sur la base de données afin d'assurer leur exactitude.

— Introduire des scripts de migration de schéma ou des bibliothèques comme ou le système de migration intégré de pour gérer les changements de schéma de manière systématique.

— Consolider la logique des requêtes SQL en utilisant des classes ou méthodes utilitaires pour réduire la duplication et centraliser les modifications.