Université de Namur

Évolution de systèmes logiciels

Rapport de projet

Vespucci

Janvier 2024

LECLERCQ Théo
MIGLIONICO Massimo
VERLY Jonah

Table des matières

1 Étape 1

Dans cette partie, il nous est demandé de faire de la **rétro ingénierie** à partir de l'analyse du repository par SQLInspect, afin d'en retrouver un **schéma physique de la base de données (PS)**, un **schéma logique (LS)** et enfin en déduire le **schéma conceptuel (CS)**.

1.1 Schéma Physique

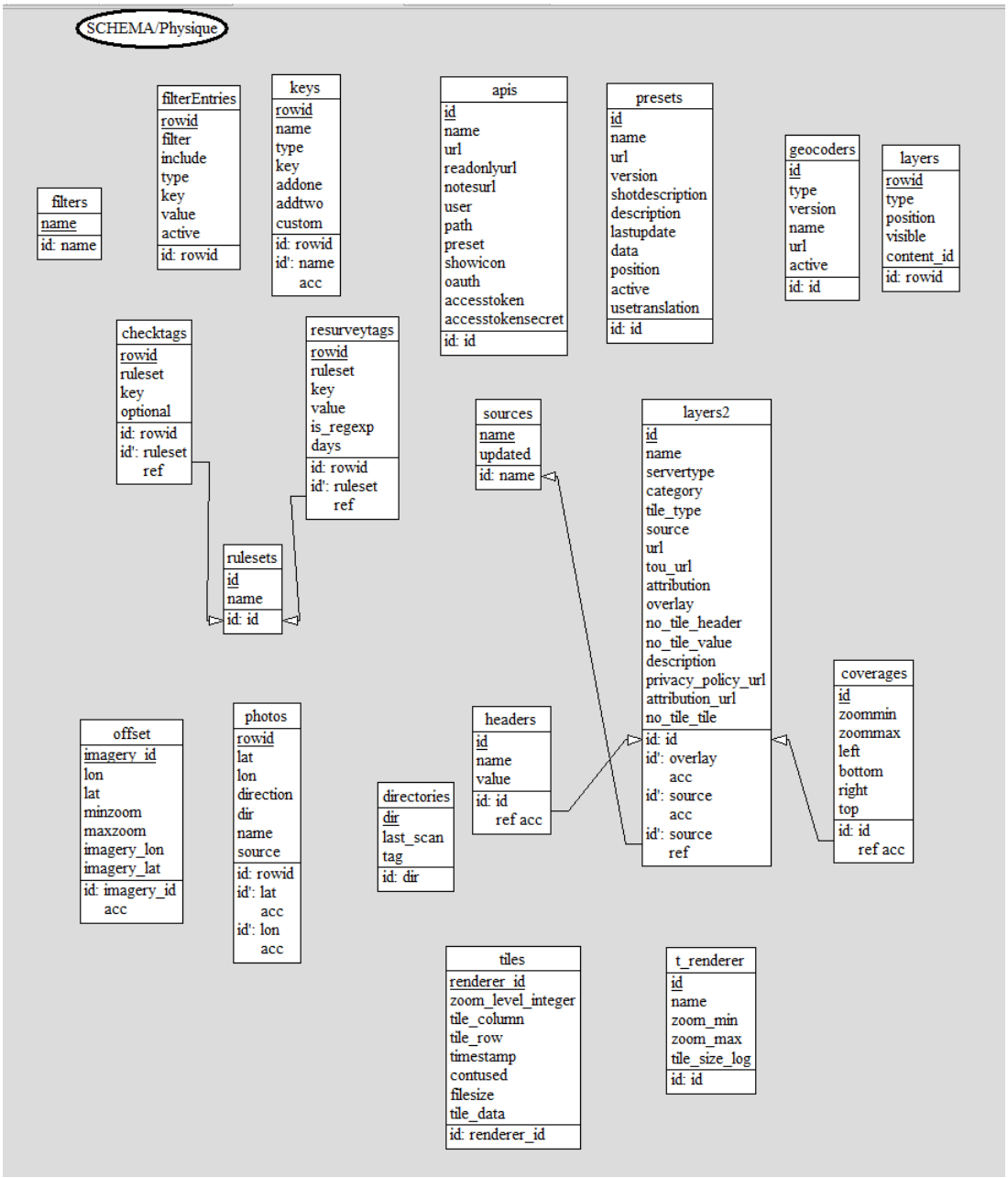

FIGURE 1 – Schéma physique de la DB

1.1.1 Processus

Pour obtenir le schéma physique, nous avons exécuté l'outil **SQLInspect** sur le code source, et ce dernier en a dérivé une liste de tables. Nous avons reconstitué les différents attributs de tables et les clés explicites à l'aide d'une analyse des traces de code. Étant donné que le code SQL a été réalisé en **SQL lite**, certaines clés primaires n'ont pas automatiquement été détectées par SQLInspect, ces clés sont générées automatiquement et ont pour libellé **rowid**. Nous les avons donc rajoutés au schéma physique.

NB : Certaines tables peuvent être supprimées lors de l'appel à `"on downgrade"`, ce qui signifie que la strucure de la base de données peut varier lors de l'exécution.

1.2 Schéma Logique

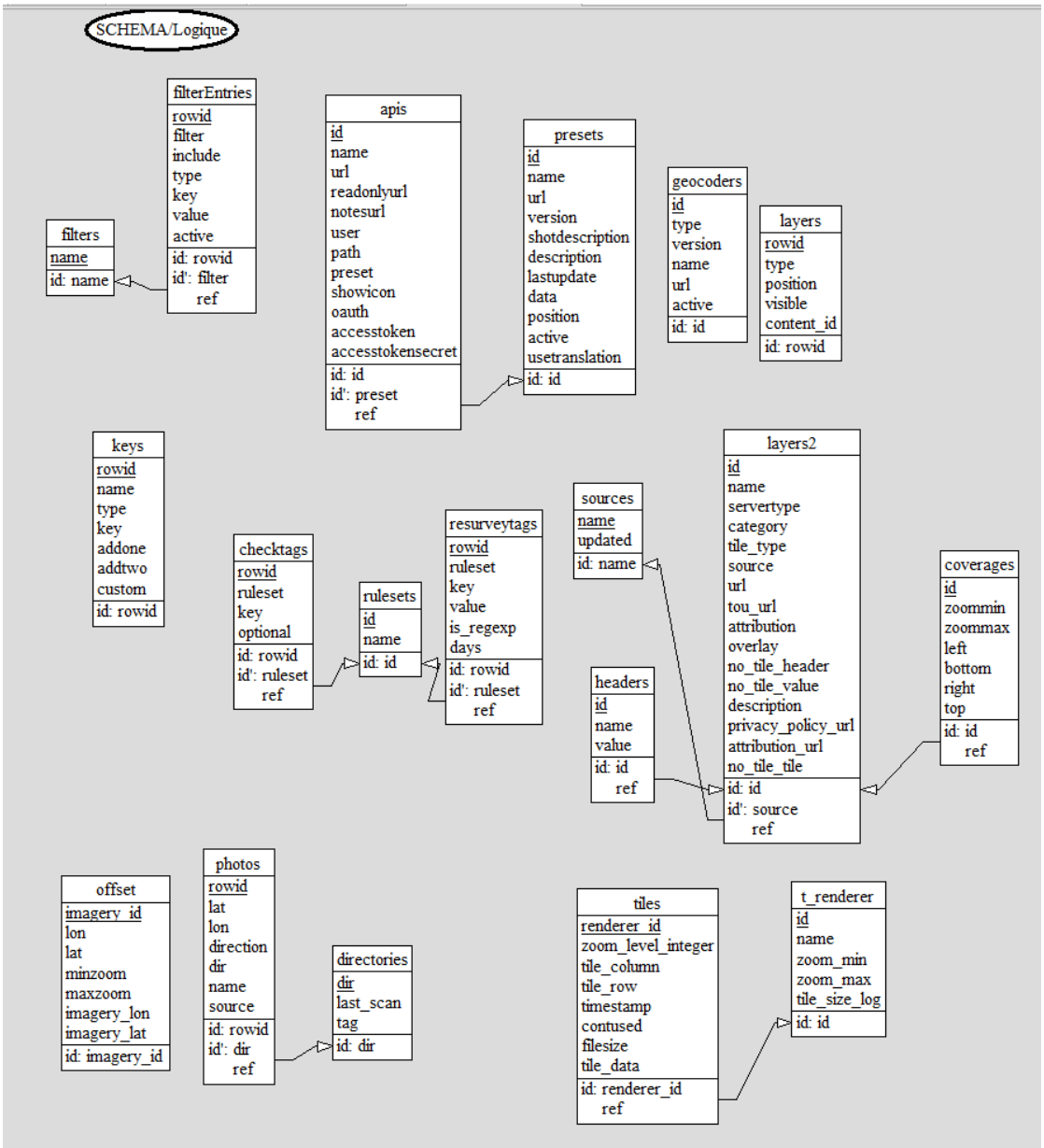

FIGURE 2 – Schéma logique de la DB

1.2.1 Processus

Pour réaliser cette étape, nous avons analysé le code source ce qui a permit d'identifier des clés étrangères supplémentaires. Cette analyse s'est de nouveau faite au travers de l'outil **SQLInspect** qui a détecté les différentes **classes java** effectuant les **requêtes SQL**. Une fois les classes connues, nous avons analysé ces classes à la main pour comprendre le comportement du logiciel par rapport aux données qu'il récupère de la base de données.

1.2.2 filters et filterEntries

Une **clé étrangère** allant de l'attribut **filter** de la table **filterEntries** vers la table **filters** a été ajoutée. Il n'y a pas de requête SQL regroupant ces deux tables. Cependant, l'attribut filter de la table filterEntries étant fort similaire au nom de la table filters nous a indiqué que ces tables sont liées. De plus en parcourant les requêtes SQL, nous avons remarqué que **ces deux tables sont créées en même temps dans la même classe java**, ce qui nous a conforté dans notre choix.

1.2.3 photos et directories

Encore une fois, ce n'est pas par une requête SQL que nous avons trouvé cette clé étrangère mais par le **nom de l'attribut dir** qui nous a fait penser à la table **directories** possédant elle aussi un attribut dir, lui-même étant la clé primaire de la table. Comme pour la clé précédente, les deux tables en question sont créées dans la **même classe java**.

1.2.4 tiles et t_renderer

De la même façon que pour les autres clés. Nous avons remarqué que la clé primaire *renderer_id* de la table **tiles** rappelle la table **t_renderer**. De nouveau ces tables sont créées dans la même classe java ce qui confirme notre hypothèse.

1.2.5 apis et presets

Cette dernière clé est encore trouvée de manière similaire au précédentes. L'attribut *preset* de la table **apis** fait référence à la table **presets**, ce qui est confirmé dans le code java qui créée ces tables au même moment.

1.2.6 Clés implicites trouvées par requêtes SQL

Diverses requêtes **SQL** nous ont indiqué des clés étrangère, les voici :

```
SELECT  coverages.id as id,left,bottom,right,top,
        coverageszoom_min as zoom_min,
        coverages.zoom_max as zoom_max
FROM layers,coverages
WHERE layers.rowid={{question}}
AND layers.id=coverages.id


SELECT headers.id as id,headers.name as name,value
FROM layers,headers
WHERE headers.id=layers.id
AND overlay={{question}}
```

```
SELECT  coverages.id as id,left,bottom,right,top,
        coverageszoom_min as zoom_min,
        coverages.zoom_max as zoom_max
FROM layers,coverages
WHERE coverages.id=layers.id
AND overlay={{question}}
```

```
SELECT resurveytags.rowid as _id, key, value, is_regexp, days
FROM resurveytags, rulesets
WHERE ruleset = rulesets.id and rulesets.name = {{question})
ORDER BY key, value
```

```
SELECT checktags.rowid as _id, key, optional
FROM checktags, rulesets
WHERE ruleset = rulesets.id and rulesets.name = (question)}
ORDER BY key
```

Ces requêtes associent respectivement layers.id à coverages.id, headers.id à layers.id, coverages.id à layers.id, resurveytags.ruleset à rulesets.id et checktags.ruleset à rulesets.id. Cependant ce colonnes sont déjà reliées par des clés étrangères explicite et ce qui signifie que ces différentes requêtes SQL ne nous ont pas apporté d'informations supplémentaires pour le schéma logique.

1.2.7 Suspicions

Nous avons trouvé plusieurs indices laissant penser que certaines tables pourraient être liées, mais par manque de preuve concrètes, nous avons décidé de ne pas créer de clé étrangères. Les voici :

- offset et photos : La table offset possède comme clé primaire imagery_id qui est un nom assez proche de la table photos. De plus, elle possède des attributs lon et lat tout comme la table photos.
- apis, presets, geocoders et layers : Ces quatres tables : Ces 4 tables sont créées dans la classe. AdvancedPrefDatabase. Elles possèdent égalements plusieurs attributs en commun tels qu'url, version, position, active. Ces noms d'attributs étant assez génériques, il nous a été impossible de définir des liens exacts entre ces tables.
- tiles et layers2 : La classe java créant la table layers2 s'appelle TileLayerDatabase. La table contient les éléments tile_type, no_tile_header, no_tile_value et no_tile_tile mais aucun de ces attributs ne se rapprochent de ceux de la classe tiles, c'est pourquoi nous ne les avons pas liées.
- keys et apis : Nous avions au départ lié la table keys au tables filterEntries, resurveytags et checktags possédant toutes trois un attribut keys. Puis nous nous sommes rendu compte que la table keys fait référence à des clés d'authentification contrairement aux autres. Une des fonctions de la classe créant la table

keys s'appelle getOAuthConfiguration et prends comme entrée un nom d'API, ce nous indique que les tables sont liées mais nous n'avons ils ne possèdent pas d'attribut similaire.

— checktags et resurveytags avec filterEntries : checktags et resurveytags sont créés et appelé dans les tables TagFilterDatabaseHelper et TagFilterActivity, ce qui les lies à filterEntries. De plus, ils ont chacun l'attribut en commun. Cependant, les requêtes SELECT sur filterEntries prennent pas en compte les valeurs de resurveytags ou checktags, c'est pourquoi nous n'avons pas créé de clés étrangères entre ces tables.

1.3 Schéma Conceptuel

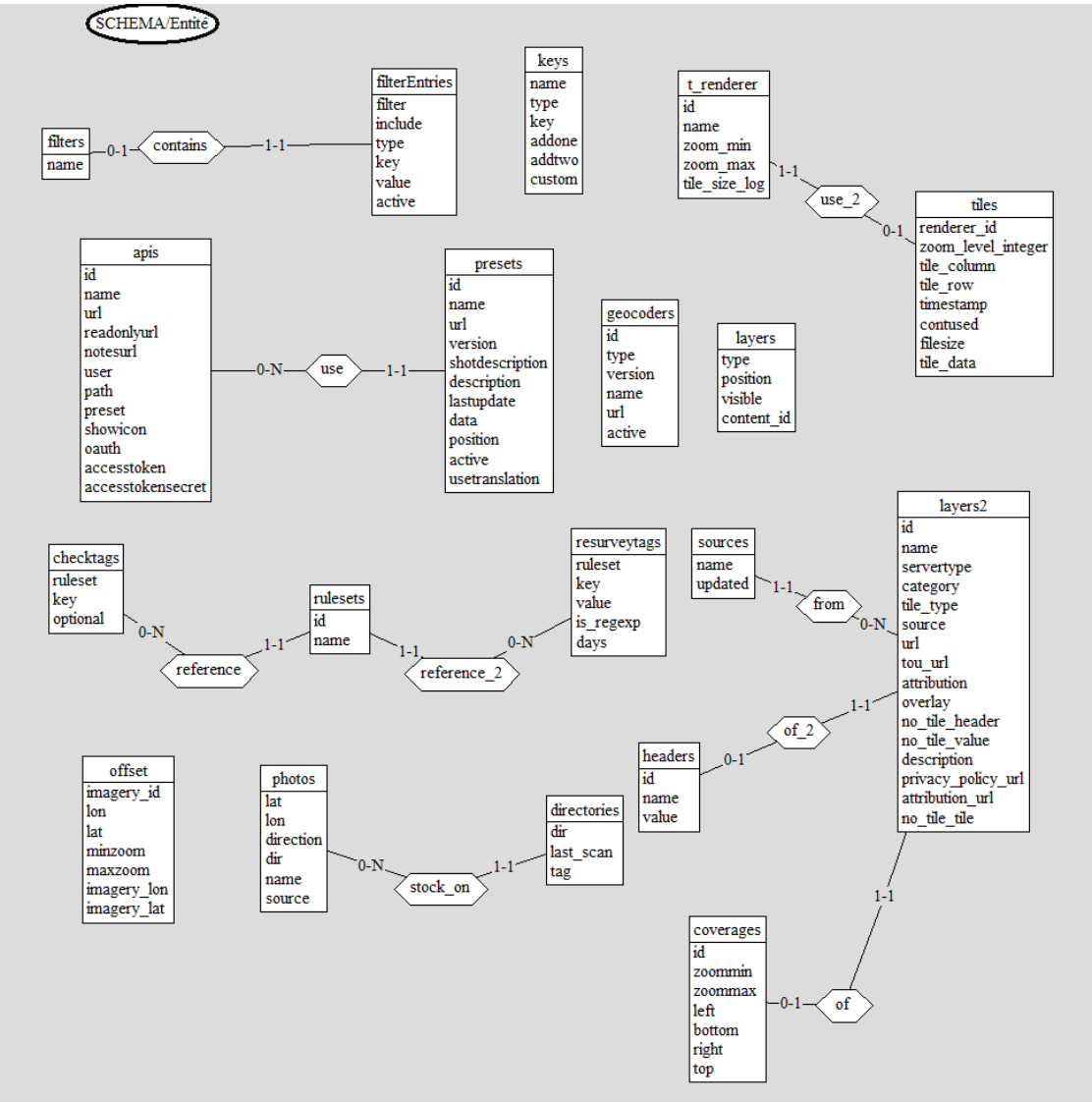

FIGURE 3 – Schéma conceptuel de la DB

1.3.1 Processus

Notre schéma conceptuel se base sur un **modèle Entité/Association**. Chaque **FK implicite ou explicite** a été remplacée par une association en fonction des tables concernées et des **cardinalités** que l'on a pu déduire. Il est important de noter que toutes les cardinalités non pas pu être déduites.

Afin de convertir celui-ci en modèle Entité/Association, il nous a fallu "retirer" les ids des tables et adapter .

1.3.2 Raisonement

Lorsque nous n'avons rien pu trouver de concret au niveau des cardinalités des FK, nous avons placé la cardinalité de **0-N** afin d'avoir la "restriction" minimum. L'association entre headers et layers nous donne une **cardinalité de 0-1** car la foreign key est également l'id de la table, ce raisonnement à également été appliqué pour les associations entre **coverage et layers, et tiles et t_renderer**.

NB :Les associations utilisant un _2 sont dues au logiciel utilisé ne permettant pas d'utiliser deux fois le même noms pour les liaisons.

2 Étape 2

Nous avons analysé les requêtes détectées par SQLInspect ainsi que le code source du projet pour nettoyer le schéma en retirant les éléments n'étant jamais utilisés, les voici :

— Table filters
— Table t_renderer
— Attribut contused de la table tiles

2.1 Sous-schéma Logique

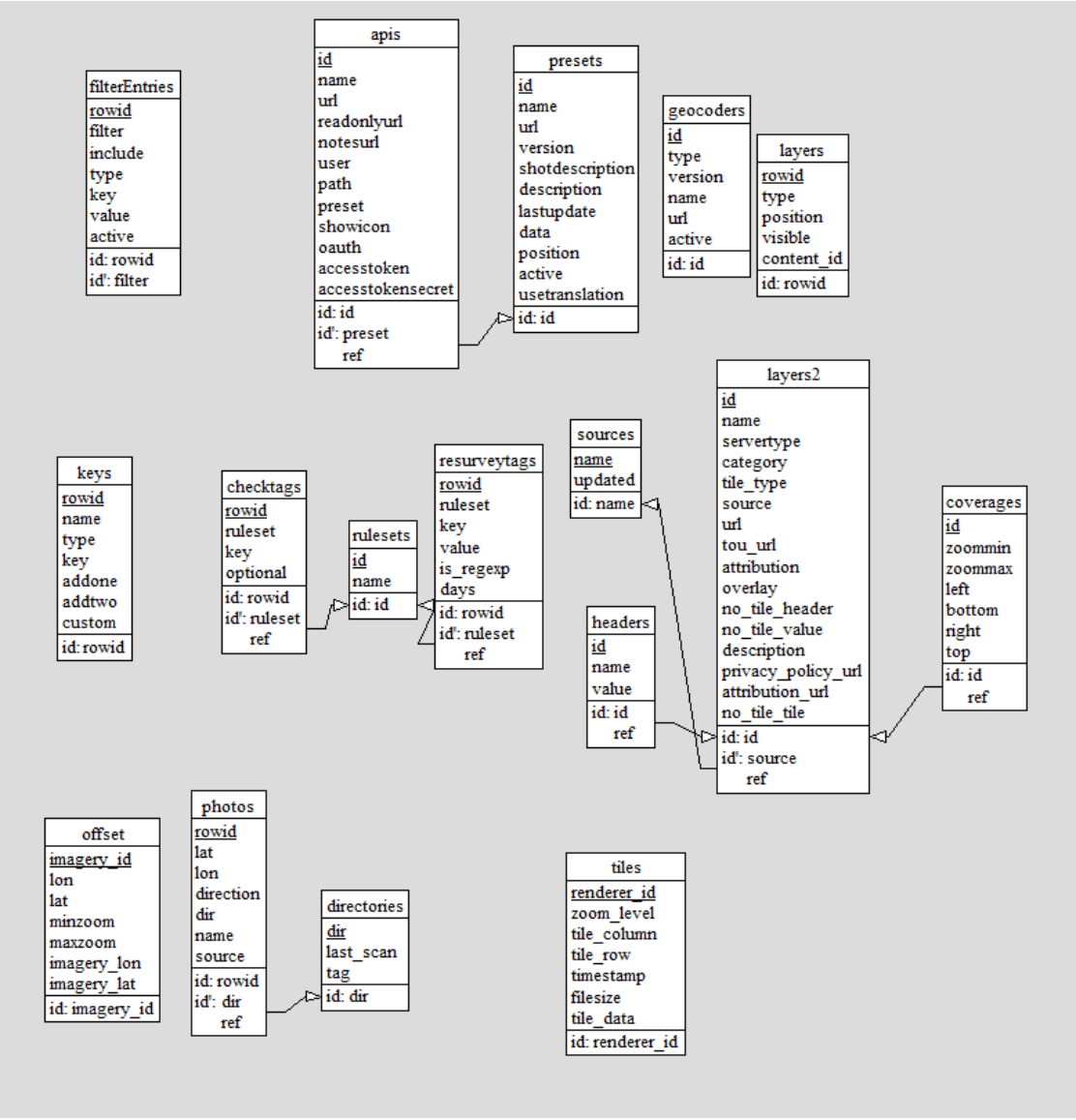

FIGURE 4 – Sous-schéma logique de la DB

2.2 Statistiques des queries

Ci-dessous, les statistiques des queries récupérés par SQLInspect.
Il est important de noter que SQLInspect n'a pas détecté toutes les fonctions manipulant la base de données. Celles détectées sont `rawQuery()` et `execSQL()`. Celles ne l'étant pas sont `query()` et `insert()` et ne sont donc pas représentées dans les tableaus ci-dessous.

TableName	select	delete	update	insert	create table	create index	drop	alter	pragma	noname	query
	0	0	0	0	0	0	0	0	1	0	0
na	0	0	0	0	0	0	0	0	0	2	2
filters	0	0	0	1	1	0	0	0	0	0	2
filterEntries	4	0	0	0	1	0	0	0	0	0	5
offsets	0	0	0	0	1	1	0	0	0	0	2
photos	0	2	0	1	2	1	2	0	0	0	8
directories	0	0	0	4	1	0	1	1	0	0	7
apis	0	0	2	0	1	0	1	6	0	0	10
presets	0	0	1	0	1	0	1	6	0	0	9
geocoders	0	0	1	0	2	0	1	0	0	0	4
layers	0	0	0	0	2	0	1	0	0	0	3
sources	0	0	0	0	1	0	0	0	0	0	1
layers2	6	0	0	0	1	2	0	8	0	0	17
coverages	2	0	0	0	1	1	0	0	0	0	4
headers	1	0	0	0	1	0	0	0	0	0	3
keys	0	0	0	0	1	1	1	0	0	0	3
tiles	4	1	0	0	1	0	0	0	0	0	6
t_renderer	0	0	0	0	1	0	1	0	0	0	2
resurveytags	3	0	0	0	1	0	0	1	0	0	5
checktags	3	0	1	0	1	0	0	0	0	0	5
rulesets	2	0	0	0	1	0	0	0	0	0	3
Total	25	3	5	5	21	8	8	24	1	2	102

TABLE 1 – Résumé des opérations de la DB

Ce deuxième tableau représente les statistiques des liens entres les tables trier par type d'associations

Tables	**Select Table 1**	**Select Table 2**	**Join**	**FK Usage**	**FK Validator**
layers2 and coverages	6	2	2	0	0
layers2 and headers	6	2	1	0	0
resurveytags and rulesets	3	2	1	0	0
checktags and rulesets	3	2	1	0	0
Total	**18**	**8**	**5**	**0**	**0**

TABLE 2 – Nombre d'opérations dans la DB pour les combinaisons de tables

3 Étape 3

3.1 Scénario 1

3.1.1 Description

Utilisation d'alter table pour la table **layers** au lieu de la supprimer et puis de la recréer.

3.1.2 Impact

Modifier la table Layers au lieu de la supprimer et puis de la recréer comprendrait moins de lignes de code DDL et faciliterait la compréhension de la base de données lors d'une mise à jour. Pour cela, la méthode `onUpgrade()` pourrait servir à recréer les lignes que l'on souhaiterait ajouter.

3.2 Scénario 2

3.2.1 Description

Création d'une table **tags** ayant une relation is_a avec les tables **checktags** et **resurveytags**.

3.2.2 Impact

resurveytags et checktags sont tous deux des sortes de tags différents ayant le même comportement : ils ont une clé étrangère vers ruleset ainsi que des fonctions fort similaires. Pour mettre ce scénario en place, il faudrait tout d'abord créer la table tags dans la classe ValidatorRulesDatabase. Ensuite, il faudrait modifier les différentes constantes contenant des requêtes SQL dans la classe **ValidatorRulesDatabase**. Cela permettrait de factoriser les requêtes comme celles-ci :

```
SELECT resurveytags.rowid as _id, key, value, is_regexp, days
FROM resurveytags, rulesets
WHERE ruleset = rulesets.id and rulesets.name = (question))
ORDER BY key, value

SELECT checktags.rowid as _id, key, optional
FROM checktags, rulesets
WHERE ruleset = rulesets.id and rulesets.name = {{question})
ORDER BY key
```

où rulesets n'interagirait plus qu'avec le table tags

3.3 Scénario 3

3.3.1 Description

Fusion des tables **filter** et **filterEntries**.

3.3.2 Impact

La table **filter** ne contenant qu'une seule colonne n'a pas de réel intérêt d'autant plus que celle-ci est atteinte par la clé étrangère de **filterEntries**. La colonne *name* pourrait donc directement être intégrée à **filterEntries**, ce qui simplifierait les requêtes `SELECT` sur **FilterEntries**, ne devant plus intégrer **filter**. La table **filters** n'étant pas utilisée, il suffirait de supprimer la requête de création de la table ainsi que la constante contenant le nom de la table dans la classe *TagFilterDatabaseHelper* pour mettre en place ce scénario.

3.4 Scénario 4

3.4.1 Description

Suppression de la table **apis**.

3.4.2 Impact

Il n'est peut-être pas nécessaire de stocker les apis dans la BD, il serait plus judicieux de les stocker dans des structures de données au travers du code. La suppression de cette table entraînerait cependant la suppression de la clé étrangère vers la table presets, il faudrait donc modifier le code source en fonction de ces changements : par exemple une classe *Apis* dans la classe **AdvancedPrefDatabase** où chaque objet pointerait vers presets. La méthode `onDowngrade()` permettrait de supprimer cette table mais il faudrait la modifier en conséquence pour ne pas supprimer les autres tables, ou bien recréer une méthode propre à la suppression de cette table. Pour recréer le lien vers la table presets, il faudrait modifier la méthode `onUpgrade()` et effectuer les changements nécessaires.

3.5 Scénario 5

3.5.1 Description

Fusion des tables **photos** et **directories**.

3.5.2 Impact

La table directories n'est utilisée que par la table photos et ne contient que 2 lignes. De plus, une photo n'est pas nécessairement stockée dans un directory, les lignes présentes dans la table directories seraient incluses dans la table photos accompagnées d'un [0-1] pour afficher leur caractère optionnel. Pour cela, il faudrait changer l'entièreté des fonctions de la classe PhotoIndex pour créer et accéder à cette nouvelle table.

3.6 Scénario 6

3.6.1 Description

Renommage d'une des tables **layers**.

3.6.2 Impact

Deux tables différentes ayant pour nom **layers** sont créées dans leur classe java respectives. Cela peut causer des conflits lors de requêtes SQL, ce qui peut être une source d'erreurs. Modifier le nom d'une des tables serait une bonne pratique et facilement réalisable étant donné que les noms des tables de la base de données sont stockés dans des constantes. Il suffira donc de changer la constante `LAYERS_TABLE` de la classe *AdvancedPrefDatabase* pour réaliser ce scénario.

3.7 Scénario 7

3.7.1 Description

Suppression d'une des tables ne possédant pas de clé étrangère.

3.7.2 Impact

L'impact de cette modification ne serait pas extensif du fait que nous n'ayons pas été en mesure de trouver une utilisation directe de cette table au travers des requêtes SQL. La classe *ImageryOffsetDatabase* serait quand même à modifier afin de mettre à jour correctement son code source.

3.8 Scénario 8

3.8.1 Description

Création d'une clé étrangère entre les tables **t_renderer** et **layers** afin d'en faciliter l'accès via la table **layers**.

3.8.2 Impact

Ce scénario vient de l'intuition que tiles/t_rendrerer est lié à layers au vu des paramètres de cette dernière table. Nous faciliterions donc la manière de retrouver les tiles liés à une couche en particulier. Nous pourrions donc rajouter une clé étrangère *renderer_id* dans **layers**.

```
"CREATE TABLE layers (id TEXT NOT NULL PRIMARY KEY,
        name TEXT NOT NULL,
        server_type TEXT NOT NULL,
        category TEXT DEFAULT NULL, tile_type TEXT DEFAULT NULL,"
        + " source TEXT NOT NULL, url TEXT NOT NULL," + " tou_url TEXT,
        attribution TEXT, overlay INTEGER NOT NULL DEFAULT 0,"
        + " default_layer INTEGER NOT NULL DEFAULT 0,
        zoom_min INTEGER NOT NULL DEFAULT 0,
        zoom_max INTEGER NOT NULL DEFAULT 18,"
        + " over_zoom_max INTEGER NOT NULL DEFAULT 4,
```

```
        tile_width INTEGER NOT NULL DEFAULT 256,
        tile_height INTEGER NOT NULL DEFAULT 256,"
        + " proj TEXT DEFAULT NULL, preference INTEGER NOT NULL DEFAULT 0,
        start_date INTEGER DEFAULT NULL, end_date INTEGER DEFAULT NULL,"
        + " no_tile_header TEXT DEFAULT NULL,
        no_tile_value TEXT DEFAULT NULL,
        no_tile_tile BLOB DEFAULT NULL, logo_url TEXT DEFAULT NULL,
        logo BLOB DEFAULT NULL,"
        + " description TEXT DEFAULT NULL,
        privacy_policy_url TEXT DEFAULT NULL,
        attribution_url TEXT DEFAULT NULL,
        FOREIGN KEY(source) REFERENCES sources(name) ON DELETE CASCADE,
        FOREIGN KEY (renderer_id) REFERENCES t\_renderer(id))");

);
```

La classe Java impactée serait *TileLayerDatabase*, et les méthodes impactées seraient `onCreate()` et `onUpdate()` avec un code SQL Lite, comme vu précédemment. Il faudrait alors faire attention lors de la création de nouvelles instances de *layers* à bien les associer aux bonnes *tiles*.

3.9 Scénario 9

3.9.1 Description

Fusion des tables **source** et **layers**.

3.9.2 Impact

Le seul lien externe de *source* que nous avons trouvé est avec la table *layers*. La première étant une table relativement petite, il serait préférable de fusionner ces deux tables en ajoutant les attributs *sourceName* et *sourceUpdated*, permettant ainsi de se passer d'une clé étrangère et d'une table supplémentaire. Cela faciliterait également les accès aux données des sources depuis la table *layers*.

```
"CREATE TABLE layers (id TEXT NOT NULL PRIMARY KEY, name TEXT NOT NULL,
        server_type TEXT NOT NULL,
        category TEXT DEFAULT NULL, tile_type TEXT DEFAULT NULL,"
        + " source_name TEXT NOT NULL,
        source_updated INTEGER,url TEXT NOT NULL,"
        + " tou_url TEXT, attribution TEXT,
        overlay INTEGER NOT NULL DEFAULT 0,"
        + " default_layer INTEGER NOT NULL DEFAULT 0,
        zoom_min INTEGER NOT NULL DEFAULT 0,
        zoom_max INTEGER NOT NULL DEFAULT 18,"
        + " over_zoom_max INTEGER NOT NULL DEFAULT 4,
        tile_width INTEGER NOT NULL DEFAULT 256,
```

```
tile_height INTEGER NOT NULL DEFAULT 256,"
+ " proj TEXT DEFAULT NULL,
preference INTEGER NOT NULL DEFAULT 0,
start_date INTEGER DEFAULT NULL,
end_date INTEGER DEFAULT NULL,"
+ " no_tile_header TEXT DEFAULT NULL,
no_tile_value TEXT DEFAULT NULL,
no_tile_tile BLOB DEFAULT NULL,
logo_url TEXT DEFAULT NULL, logo BLOB DEFAULT NULL,"
+ " description TEXT DEFAULT NULL,
privacy_policy_url TEXT DEFAULT NULL,
attribution_url TEXT DEFAULT NULL)");
```

La classe Java impactée serait *TileLayerDatabase*, et les méthodes impactées seraient `onCreate()` et `onUpdate()` avec un code SQLite. Cela ne demanderait donc pas beaucoup d'adaptation du code.

3.10 Scénario 10

3.10.1 Description

Suppression d'une des colonnes de **Presets** : *ShortDescription*.

3.10.2 Impact

Cette colonne ne semble pas indispensable car faisant doublon avec la colonne *description*. Elle ne semble pas être utilisée dans les requêtes également. Le code touché par cette modification serait la classe *AdvancedPrefDatabase*, ainsi que les méthodes `onCreate()`, `getPresets()`, `getActivePresets()`, et `getPreset()`.

4 Étape 4

4.1 Base de Données

4.1.1 Points Positifs

- Utilisation d'une technologie adaptée à son support : Utiliser SQLite est une bonne idée pour un support comme un smartphone, car cela permet une base de données légère.
- Utilisation de clés étrangères explicites : Il s'agit d'une bonne pratique.

4.1.2 Points Négatifs

- Noms de table et d'attribut similaires : Confusion entre les attributs *'key'* de **'resurveytag'**, **'checktag'**, et **'filterEntries'** non liés à la table **'Keys'** ; Existence de deux tables **'layers'** différentes.
- Table non nécessaire : **'filters'** n'ayant que *'name'* comme attribut et liée à **'filterEntries'**.
- Possibilité de création de sur-type de classe : scénario 2.
- Peu de clés primaires explicites : Plusieurs tables se contentent d'utiliser l'id automatique SQLite : **rowid**, ce qui pourrait prêter à confusion.

4.2 Code

4.2.1 Points Positifs

- Commentaires : Chaque classe et méthode est accompagnée de commentaires explicatifs.
- Utilisation de constantes pour les requêtes.

```
final Cursor c = mDatabase
        .rawQuery(
                "SELECT " + T_FSCACHE_ZOOM_LEVEL + "," + T_FSCACHE_TILE_X + "," + T_FSCACHE_TILE_Y + "," + T_FSCACHE_FILESIZE + " FROM "
                        + T_FSCACHE + " WHERE " + T_FSCACHE_RENDERER_ID + "='" + rendererID + "' ORDER BY " + T_FSCACHE_TIMESTAMP + " ASC",
                null);
```

FIGURE 5 – Exemple de constante

```
static final String QUERY_RESURVEY_DEFAULT  = "SELECT resurveytags.rowid as _id, key, value, is_regexp, days FROM resurveytags WHERE ruleset = "
        + DEFAULT_RULESET + " ORDER BY key, value";
```

FIGURE 6 – Exemple de constante

4.2.2 Points Négatifs

- Documentation insuffisante ou peu claire : Documentation réduite et commentaires souvent insuffisants pour une compréhension complète.

- Paramètres et constantes non explicités : Utilisation d'attributs *'key'* dans **'resurveytag'** sans explication claire, rendant l'objectif de l'attribut *'key'* ambigu.
- Code peu compréhensible.

4.3 Méthodes d'amélioration

4.3.1 Base de Données

- Noms de table et d'attribut similaires : Assurer des noms distincts et non ambigus pour chaque élément.
- Table inutile : Éviter de créer des tables avec un seul attribut ou les fusionner avec celles qui les utilisent.
- Possibilité de création de sur-type de classe : Voir scénario 2.
- Peu de clés primaires explicites : Déclarer explicitement les id dans le code et veiller à ce que leurs noms soient évidents.

4.3.2 Code

- Documentation insuffisante ou peu claire : Maintenir des schémas de BD à jour et une documentation claire, évoluant en parallèle du projet.
- Paramètres et variables peu explicités : Clarifier l'utilisation dans la documentation et les commentaires.
- Code peu compréhensible : Suivre les conventions de codage, éviter les classes trop longues (exemple : une classe de plus de 1300 lignes), et fractionner le code en nommant judicieusement les différents éléments.