# INFOM218 : Evolution de systèmes logiciels :

# Rapport groupe 1 sur Conversations

*Simon Loir, Gaëtan Ramack, Johan Rochet*


Université de Namur
Faculté d'informatique
Année académique 2023-2024

# Table des matières

# 1 Étape 1 : Extraction du schéma physique, logique & conceptuel

## 1.1 Extraction du schéma physique explicite

Nous avons commencé notre extraction par une analyse manuelle du dépôt dans un IDE afin de pouvoir découvrir la structure du projet. Grâce à cela, nous avons pu déterminer que Conversations utilise SQLite comme système de base de donnée et que l'ensemble des requêtes se faisaient dans un nombre relativement faible de fichiers. Etant donné le fonctionnement de SQLite, nous avons eu l'idée d'introduire du code afin de pouvoir obtenir des logs de la création de la base de données lors de l'exécution de l'application. Nous avons ainsi pu récupérer les requêtes *create* de l'application et les avons utilisées afin de créer un premier schéma physique. L'ensemble des *create* effectuées afin de de générer ce diagramme se trouvent dans le fichier *schema.sql* dans la step1 sur le repository.

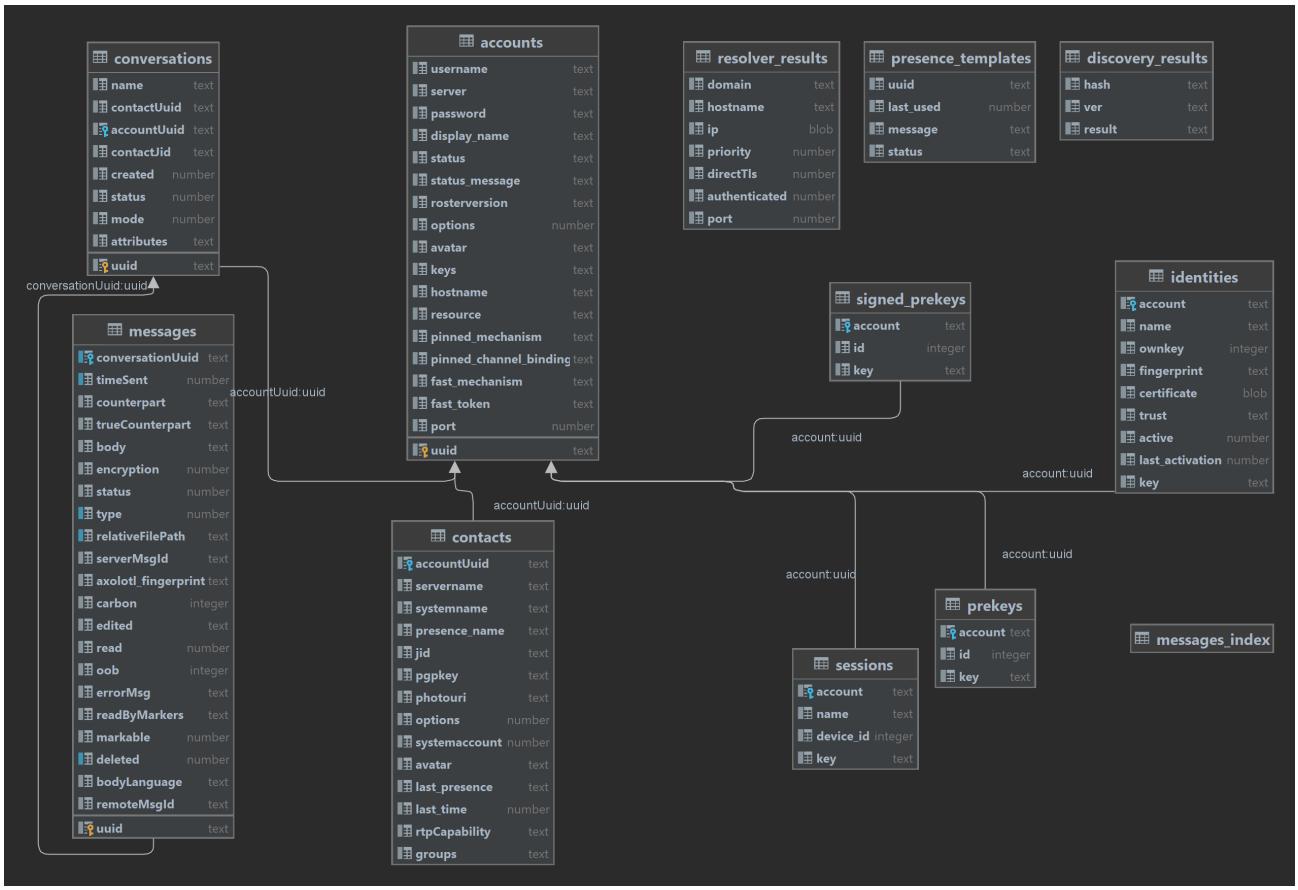


FIGURE 1 – Schéma physique explicite de la base de données

## 1.2 Extraction du schéma logique

Le schéma physique explicite ayant été récupéré, nous avons pu entamer l'analyse logique de celui-ci.

### 1.2.1 Analyse du schéma

Il est à noter que les noms de tables et colonnes ne nécessitaient pas de réflexion trop poussée afin de pouvoir inférer leur logique, mais comprenaient certaines irrégularités au niveau de la nomenclature, ce qui a engendré certaines confusions lors de la tentative de compréhension de certaines requêtes. En effet, les noms des colonnes dans la base de données font tantôt usage de Camel case, tantôt de Snake case au sein d'une même table.

Il est ressorti de l'analyse du schéma 2 grands points de questionnement à investiguer :

1. Certaines tables (*resolver_results, presence_templates et discovery_results*) ne sont reliées à aucune autre

table de la base de données

2. La table *conversations* n'est pas relié par une clé étrangère à la table *contacts* alors qu'il y a un attribut *contactUuid* et *contactJid*

### 1.2.2 Analyse des queries avec SQLInpsect

Afin de mieux comprendre les relations en jeu, nous avons procédé à une analyse des queries à la base de données. Pour ce faire, nous avons utilisé SQLInspect afin de générer les queries à la base de données. Le résultat de l'analyse avec SQLInspect se trouve dans le fichier *SQLInspect.xml* de la step 1 sur le repository.

Nous avons récupéré l'ensemble des queries repérées par SQLInspect et les avons exportées en XML pour ensuite les réécrire dans un fichier SQL (*sql-inspect-output.sql*.

De prime abord, SQLInspect confirmait nos logs initiaux de la base de données en nous générant l'ensemble des requêtes *create* que nous avions trouvées. De plus, il a su repérer une quinzaine de queries de type *select*, *insert* ou *delete*. L'analyse de ces requêtes ne nous a pas permis d'inférer facilement l'existence de nouvelles clés étrangères possibles entre certaines tables. Toutes les jointures réalisées ne se font qu'entre des clés étrangères déjà existantes dans la base de données.

De plus, le nombre de queries récupérées par l'analyseur était assez faible et ne concernait pas toutes les tables de la base de données, ce qui laissait présager que beaucoup de queries n'ont pas été détectées par ce-dernier. Une analyse manuelle plus fine du code était nécessaire afin de mieux comprendre le fonctionnement des tables et les liens qu'elles entretenaient.

### 1.2.3 Analyse du programme

Afin d'obtenir une compréhension accrue de la base de données, nous avons analysé le fonctionnement du programme afin de repérer l'existence ou non de liens entre différentes tables. Comme certaines requêtes semblaient n'avoir pas été détectées par SQLInspect alors qu'elles étaient bel et bien utilisées par l'application, nous avons exécuté l'application avec un débogueur afin de pouvoir retrouver ces requêtes et d'analyser nos questionnements a priori.

#### Une clé étrangère entre *contact* et *conversations* ?

Après analyse, nous avons remarqué que la colonne *contactUuid* de la table *contacts* n'est à aucun moment utilisée (toujours *null*). La potentielle utilité de cette colonne pour faire un lien avec la table *contacts* est en fait gérée par la colonne *contactJid*. Afin de pouvoir modéliser ce lien, nous avions, dans un premier temps, inséré une table intermédiaire jabber avec comme attribut le jid.

Toutefois, en analysant le code plus en profondeur, notamment avec le débogueur, nous avons découvert que le lien que nous avions précédemment rajouté entre la table *contacts* et *conversations* ne pouvait se faire étant donné qu'un utilisateur $A$ peut avoir une conversation avec un autre utilisateur $B$ même si $B$ n'est pas répertorié dans la liste de contact de $A$. En effet, lorsque $B$ crée la conversation et que $A$ est répertorié dans la liste de contacts de $B$, une conversation existe, mais sans que $B$ n'apparaisse dans les contact de $A$. Il semblerait aussi que les jid aient vu leur stratégie d'emploi remaniée lors de précédentes migrations.

Ainsi, on ne peut pas conclure qu'il existe un lien entre les deux tables, bien que le JID présents dans la table *conversations* se rapportent à celui de la table *contacts* dans certain cas.

#### Des tables isolées ?

Dans un premier temps, nous avons analysé la table *resolver_results*. De prime abord, aucune colonne de cette table ne semblait être liée à une quelconque autre colonne. Cependant, l'analyse du code a fait ressortir que La

table *resolver_results* et la table *accounts* entretiennent bel et bien un lien afin de pouvoir trouver un serveur mXmpp dans une liste de serveurs. Ce lien n'étant pas encore répertorié, nous avons procédé à l'ajout de la table intermédiaire *servers* contenant l'attribut *domain* afin de corriger cela.

Ensuite, nous nous sommes penchés sur la table *presence_templates*, qui grâce à son nom devrait servir à stocker des templates de statut (online, offline...) au sein de l'application. Intuitivement, cette table devrait être reliée à la table *accounts* afin de pouvoir stocker l'état de présence d'un compte sur l'application. Cependant, ce lien n'existe pas, car la table *presence_templates* semble être liée à une fonctionnalité qui n'est soit pas encore terminée, soit a été abandonnée. Actuellement, la présence est directement stockée dans la table *accounts* dans les colonnes *status* et *status_message* (colonnes similaires à *message* et *status* de la table *presence_templates*).

### 1.2.4 Schéma logique de l'application

Le schéma physique de l'application a été modifié de manière incrémentale pendant l'analyse afin d'obtenir un schéma logique final à la fin de celle-ci et pouvoir vérifier que les nouvelles structures (tables, clés étrangères) découvertes restaient cohérentes tout au long de l'analyse.

Voici le schéma logique résultant de cette analyse :

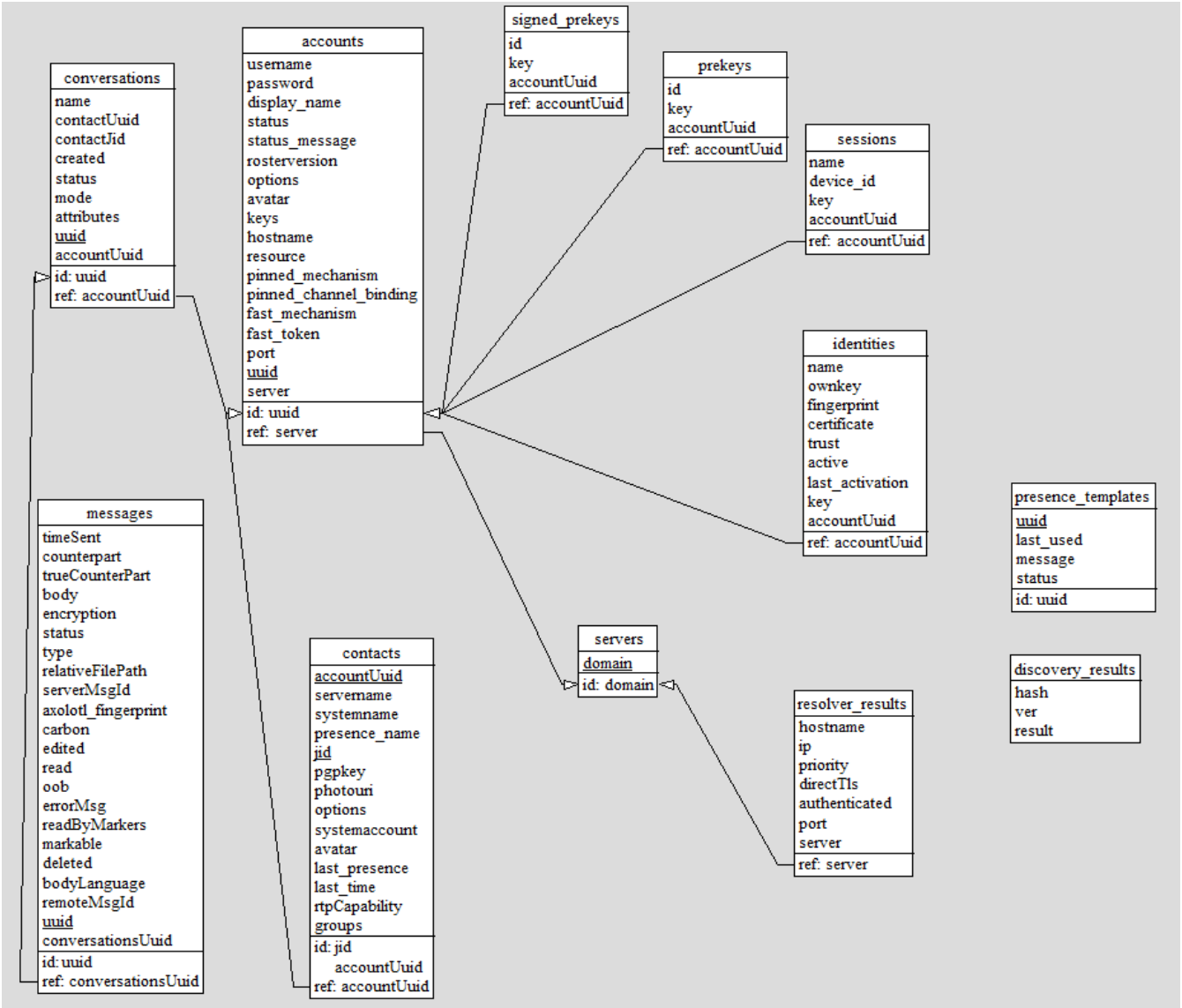


FIGURE 2 – Schéma logique de la base de données

## 1.3 Conceptualisation

Notre schéma conceptuel, élaboré à l'aide de l'outil DBMain est le suivant :

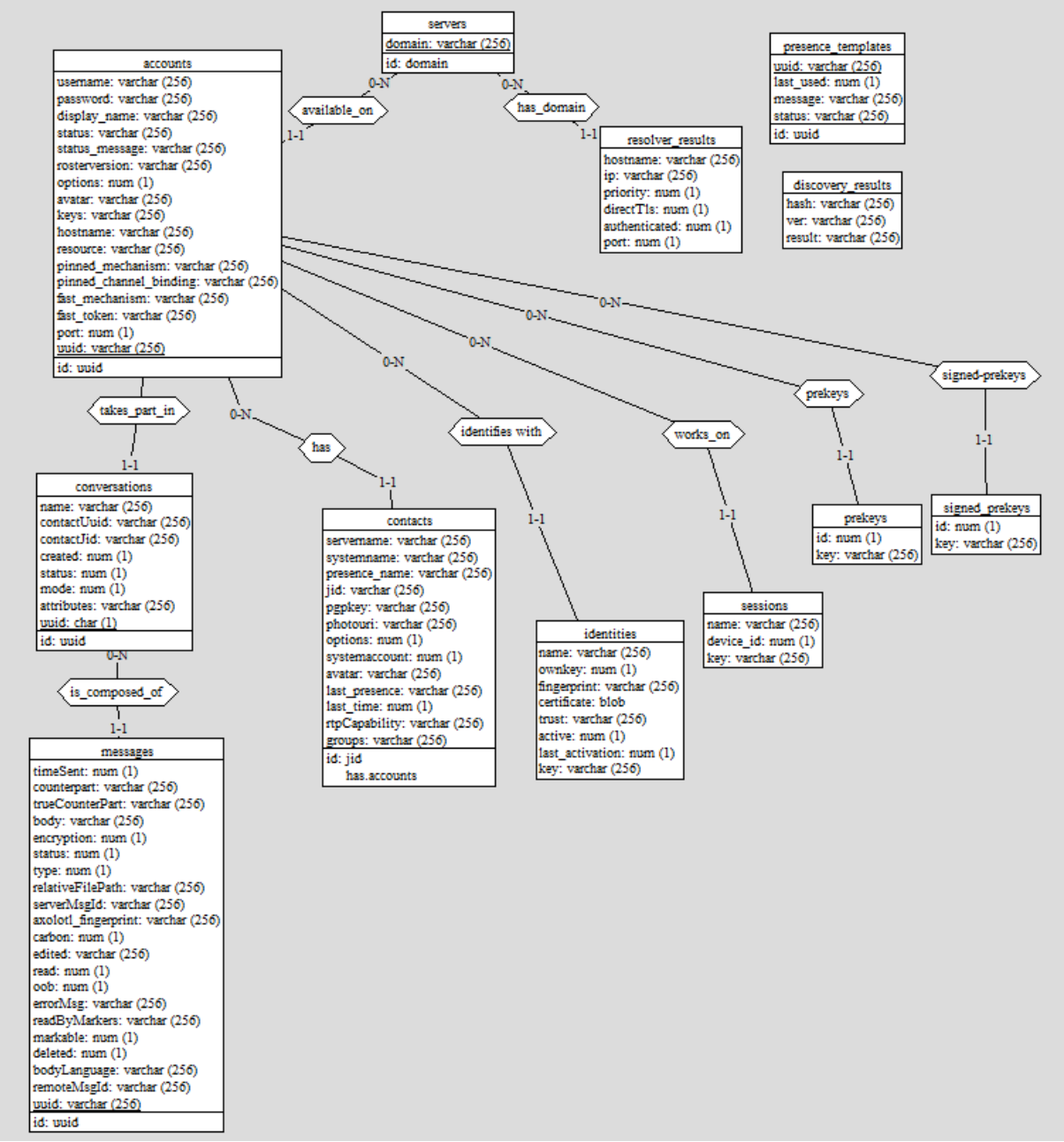


FIGURE 3 – Schéma conceptuel de la base de données

Celui-ci a pu être élaboré directement depuis le schéma logique sans complications.

Le schéma a été remanié tout au long du processus de recherche afin que celui-ci reste conforme avec nos trouvailles.

# 2 Étape 2 : Dérivation du Sublogical schema

Dans cette étape, nous cherchons à déterminer quelles parties du schéma sont réellement utilisées. Le schéma logique utilisé par l'application à un moment $t$ peut en effet contenir des tables ou des colonnes qui ne sont pas ou plus utilisées mais toujours présents dans la base de données.

## 2.1 Quelles tables sont utilisées par l'application ?

Dans un premier temps, nous cherchons à définir quelles sont les tables qui sont utilisées par l'application. Pour ce faire, l'idée initiale est de récupérer les queries qui sont faites à la base de données à l'aide de SQLInspect (analyseur **statique**) et d'en inférer l'utilisation ou non d'une table. Des requêtes SQL récupérées, nous avons pu déjà repérer que les tables :

— contacts,
— conversations,
— messages,
— accounts,
— discovery_results

étaient query avec des requêtes *select*. Cependant, et partant du constat de l'étape 1 que SQLInspect ne permettait pas de récupérer toutes les queries, faites à la base de données (notamment, aucune requêtes *insert*), nous avons mis en place un analyseur des logs **dynamiques** de la DB.

Afin de pouvoir récupérer les logs d'une exécution dynamique de la base de données, nous avons utilisé l'outil de ligne commande *Logcat* nous permettant de récupérer les logs des *SQLLiteStatements*.

Pour ce faire, nous avons tapé la commande suivante dans le shell :

```
adb shell setprop log.tag.SQLiteStatements VERBOSE
```

Cette commande nous a permis de récupérer les logs de l'applications en direct.

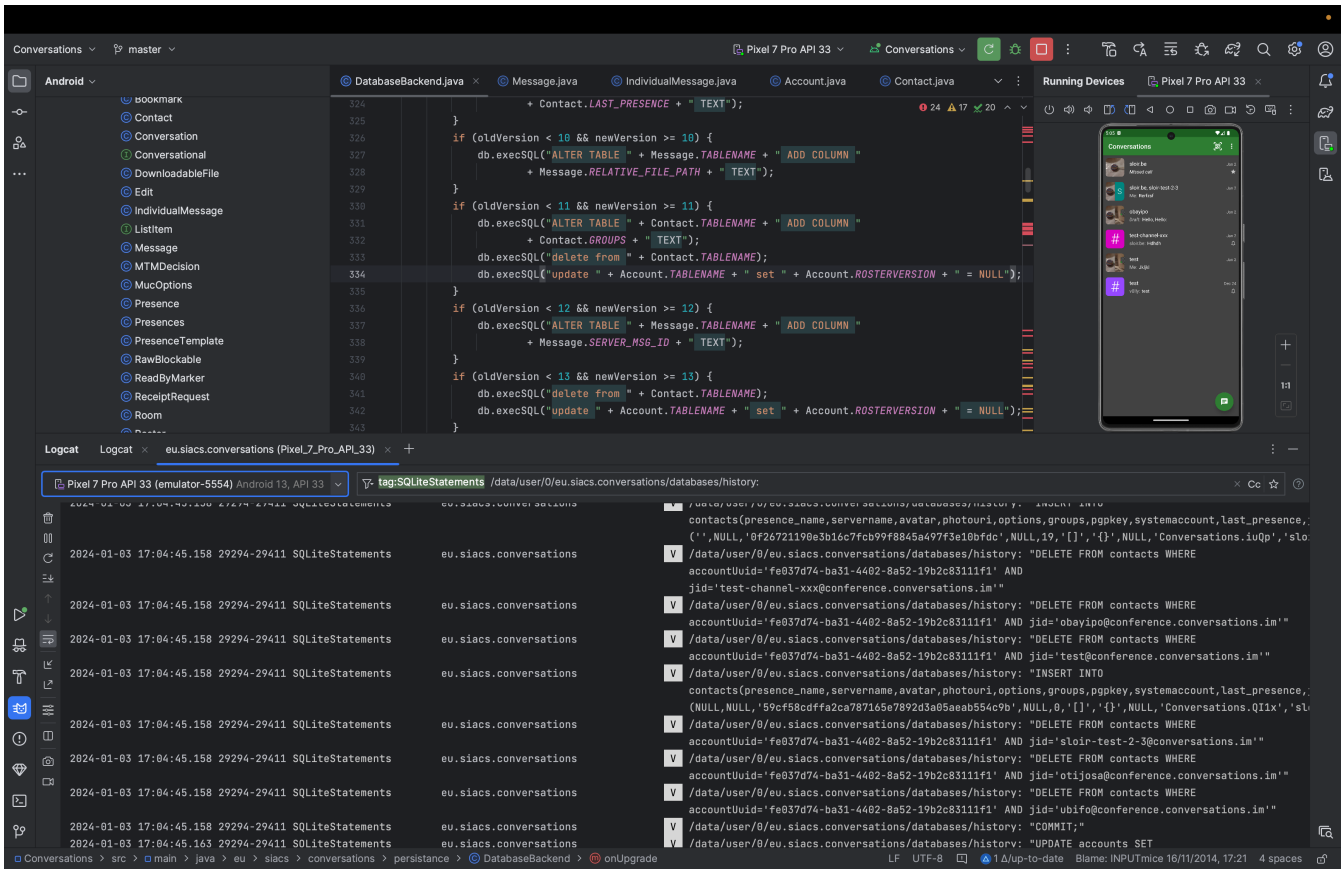


FIGURE 4 – Logs d'une exécution dynamique de l'application

Nous avons ainsi exécuté une série d'actions dans l'application afin de couvrir au maximum l'utilisation de toutes les tables. Pour analyser ces logs, nous avons mis en place un extracteur custom de requêtes SQLLite afin de pouvoir compter le nombre de requêtes *insert*, *select*, *update* et *delete* pour chaque table. Le log complet des queries faites par l'application se trouvent dans le fichier *step2/queries-log.sql* et le résultat de l'analyse dans le fichier *step2/usage.json*.

```
Map(12) {
  'messages' => Map(3) { 'select' => 18, 'update' => 6, 'insert' => 2 },
  'conversations' => Map(3) { 'update' => 14, 'select' => 13, 'insert' => 2 },
  'accounts' => Map(4) {
    'update' => 122,
    'select' => 4,
    'insert' => 1,
    'delete' => 8
  },
  'resolver_results' => Map(1) { 'select' => 4 },
  'discovery_results' => Map(1) { 'select' => 4 },
  'signed_prekeys' => Map(2) { 'select' => 6, 'insert' => 1 },
  'prekeys' => Map(2) { 'select' => 200, 'insert' => 100 },
  'sessions' => Map(2) { 'select' => 13, 'insert' => 1 },
  'contacts' => Map(3) { 'insert' => 22, 'delete' => 20, 'select' => 1 },
  'identities' => Map(3) { 'select' => 8, 'update' => 1, 'insert' => 1 },
  'presence_templates' => Map(3) { 'select' => 1, 'delete' => 2, 'insert' => 1 },
  'android_metadata' => Map(2) { 'create' => 1, 'select' => 1 }
}
Map(80) {
```

FIGURE 5 – Nombre des requêtes pour chaque table

De cette analyse, nous voyons bien que l'ensemble des tables semblent être, à un moment ou un autre, utilisés par l'application. Seul petit bémol, la table *presence_template* semble fort peu utilisée, mais nous y reviendrons dans la suite de l'analyse.

## 2.2 Quelles colonnes ne sont pas utilisées par l'application ?

Dans un second temps, nous cherchons à déterminer quelles colonnes sont réellement utilisées par le programme.

Pour ce faire, analyser chaque requêtes, faites par le programme, récupérés l'ensemble des colonnes récupérées pour chaque table dans chaque requête, semble fastidieux et peu optimal. En effet, beaucoup de requêtes récupèrent l'entièreté ou une partie des colonnes d'une table et rien ne permet de définir si le programme les utilise réellement. C'est pourquoi, il était nécessaire de pouvoir récupérer, dynamiquement, d'un côté, de l'information sur quelle colonne était utilisée dans la base de données et, de l'autre, l'utilisation que le programme en faisait.

L'*app inspection* fournie par *Android Studio*, nous a permis d'avoir accès en temps réel au contenu de la base de données de l'application. Nous avons ainsi, pu analyser les résultats des actions effectuées sur l'application directement dans la base de données et avons tenté de repérer quels colonnes ne semblaient pas être utilisées par l'application. Dans la suite, nous détaillons notre raisonnement pour chaque colonne que nous avons jugé candidate à une inspection plus précise du programme.

### 2.2.1 Dans la table *accounts*

La table *accounts* est la table centrale du programme, permettant de stocker toutes les informations sur le compte d'un utilisateur et de lier les autres tables ensemble. Cependant, deux colonnes semblaient ne pas être utilisées par l'application au runtime.

1. **Colonne : *status_message***

   Comme observé dans la première partie, la colonne *status_message* semble être liée au système de

statut de présence de l'application et aurait la même utilité que la colonne *message* dans la table *presence_templates*. Lors de l'utilisation normale de l'application, cette colonne ne semblait pas être utilisée par le programme. Toutefois, après analyse du code et recherche dans l'application, nous avons remarqué qu'en activant une fonctionnalité dans les paramètres *experts* de l'application, cette colonne permettait de stocker le message de statut si celui-ci était un message de statut par défaut proposé par l'application.

2. **Colonne : *pinned_channel_binding***

La colonne *pinned_channel_binding* s'est révélée inchangée quelles que soient nos actions bien que le programme ne comporte des fonctions et méthodes permettant de mettre à jour son état. Elle semble, en effet être utilisée dans le mécanisme de connexion . Par conséquent, nous ne pouvons déterminer avec certitude si son inclusion dans le sublogical schema est pertinente.

#### 2.2.2 Dans la table *contacts*

Dans la table *contacts*, la colonne *groups* n'est jamais mise à jour par l'application. Cependant, le code du programme semble permettre de l'utiliser, mais son utilisation n'est pas claire carla colonne est supposée stockée des arrayJson. Difficile dès lors de l'exclure totalement du sublogical schema vu que les exécutions dynamiques effectuées ne prouvent pas que, dans tous les cas, la colonne n'est pas modifiée.

#### 2.2.3 Dans la table *conversations*

Comme identifié précédemment, la colonne ***contactUuid*** semble en plus être utilisée par l'application, remplacée par la colonne *contactJid*. Notre analyse a pu confirmer cette intuition initiale et nous avons supprimé la colonne du sublogical schema.

### 2.3 Cas particulier : la table *presence_templates*

Comme évoqué précédemment, la table *presence_template* n'est pas réellement utilisée par le programme. Celui-ci insert des lignes pour chaque nouveau statut d'un utilisateur, mais jamais, il n'est proposé à un utilisateur de choisir parmi un panel de template de message de statut. Son utilité au sein du programme est ainsi discutable. notre hypothèse principale est que la fonctionnalité n'a jamais été terminée ou bien a simplement été abandonnée en cours de routes par les développeurs, car le back-end est partiellement prêt à l'utiliser, mais pas le front-end de l'application.

#### 2.3.1 Sublogical schema

Ainsi, le Sublogical schema suivant résume bien l'analyse faite. Les colonnes marquées en rouge, sont les colonnes du schéma logique à supprimer et celles marquées en orange, sont celles pour lesquels il y a encore un questionnement qu'il faudrait pouvoir régler en discutant avec l'équipe de développement afin d'être sûr qu'elles ne sont plus utilisées par l'application.

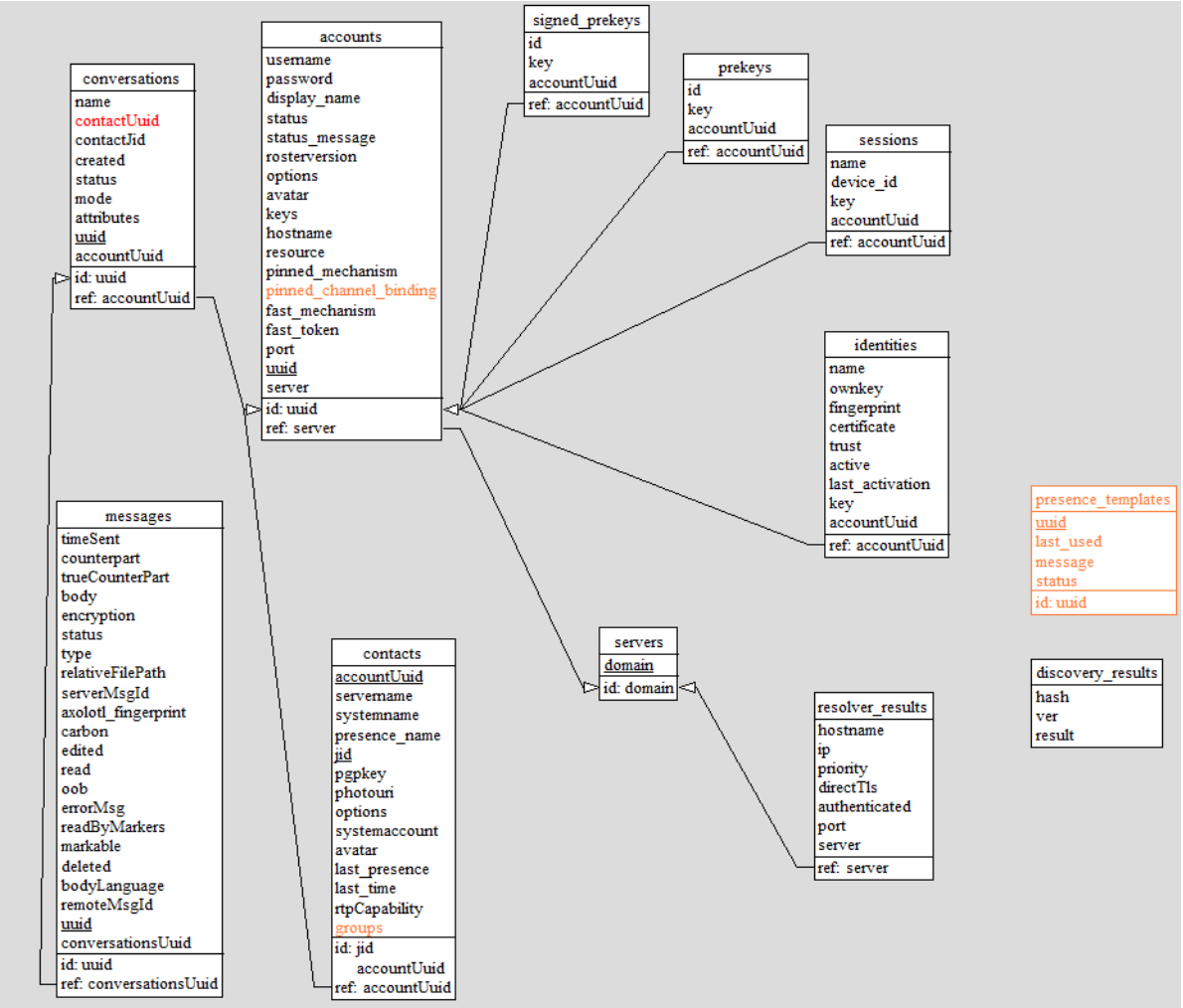


FIGURE 6 – Sublogical schema

# 3 Étape 3 : Scénarios d'évolution

## 3.1 Scenario 1 : Ajout de colonne dans la table *accounts*

Le fait que toutes les requêtes d'écriture et de lecture se font dans la classe *DatabaseBackend* et que les données relatives à ces requêtes sont directement générées au sein des entités facilite l'ajout de colonnes.

Pour ajouter une colonne dans la table *accounts*, il faut modifier la requête SQL exécutée dans la méthode *onCreate* du *DatabaseBackend*. De cette manière, la colonne est ajoutée à la création de la table. Il faut aussi ajouter une migration dans la méthode *onUpgrade* du *DatabaseBackend* et modifier la *DATABASE_VERSION* pour que ces changements soient appliqués.

Pour respecter les conventions déjà établies au sein du projet, il faudrait créer une constante *Account.NOM_DE_COLONNE* dans la classe *Account*. Il faudrait aussi créer une nouvelle propriété privée dans la classe *Account*, accompagnée d'une modification du constructeur, ainsi que l'ajout d'un setter et un getter pour accéder à cette nouvelle propriété. La méthode *fromCursor* devrait être modifiée pour prendre en compte le changement du constructeur et la méthode *getContentValues* devrait aussi être adaptée.

## 3.2 Scenario 2 : Suppression de colonne *contacts.last_presence*

Pour supprimer la colonne *last_presence* de la table *contacts*, il faudrait modifier le *CREATE_CONTACT_STATEMENT* dans le *DatabaseBackend*. Le constructeur de la classe *Contact* devra être adapté en retirant le paramètre *presence* et la méthode *fromCursor* devra être adaptée pour refléter ce changement. La propriété privée *mLastPresence* est utilisée par le setter *setLastResource* et *getLastResource*. Cependant, en ajoutant un break point avec le débogueur, il semblerait que *setLastResource* ne soit pas appelé lors de l'exécution bien qu'il soit appelé dans le classe *AbstractParser*. Par ailleurs, *getLastResource* semble être utilisée dans la classe *PresenceSelector*, dans la méthode *showPresenceSelectionDialog* qui devrait être adaptée.

## 3.3 Scenario 3 : Suppression de la table *presence_templates*

Dans le cas où la décision serait prise de supprimer définitivement la table *presence_templates*, peu de modifications du code seraient nécessaires. Il faudrait :

1. Supprimer la table du fichier *DatabaseBackend.java*
2. Rajouter une migration pour une nouvelle version de la base de données dans le fichier *DatabaseBackend.java* et supprimer les méthodes pour insérer dans cette table.
3. Supprimer la partie de code concernant l'insertion dans la table dans le fichier *XmppConnectionService.java*.
4. Supprimer le code de l'entité *PresenceTemplate* qui ne sera plus utilisé (pour éviter d'avoir du code mort dans l'application).

```
public void changeStatus(Account account, PresenceTemplate template, String signature) {
    if (!template.getStatusMessage().isEmpty()) {
        databaseBackend.insertPresenceTemplate(template);
    }
    account.setPgpSignature(signature);
    account.setPresenceStatus(template.getStatus());
    account.setPresenceStatusMessage(template.getStatusMessage());
    databaseBackend.updateAccount(account);
    sendPresence(account);
}
```

FIGURE 7 – Code insertion de données dans la table *presence_templates*

### 3.4 Scenario 4 : Renommage de colonne *contactJid* en *contact_jid*

Afin de pouvoir uniformiser la politique de nomenclature en snake_case, nous avons décidé de renommer la colonne *contactJid* de la table *conversation*. Cette colonne est utilisée afin de récupérer les différentes conversations liées à un contact. Étant donné que l'application se base sur un pattern DAO, le renommage de cette colonne ne nécessiterait que de renommer les instances de *CONTACTJID* rencontrées dans le fichier *DatabaseBackend.java* ainsi que de renommer *CONTACTJID* dans la classe *Conversation*.

### 3.5 Scenario 5 : Merge de *signed_prekeys* et *prekeys* en *prekeys*

On pourrait centraliser le stockage des clés dans une seule table nommée "prekeys" puisque les tables *signed_prekeys* et *prekeys* ont les mêmes colonnes.

Pour ce faire, il faudrait ajouter une colonne "signed" à la table *prekeys* et supprimer la table *signed_prekeys*. Dans le DatabaseBackend, il suffirait alors de fusionner les méthodes *getCursorForSignedPreKey* et *getCursorForPreKey* puisqu'elles ont déjà un code qui est similaire. Il faudrait alors ajouter un paramètre "signed" booléen et ajouter une condition à la clause where de la requête.

Ces changements auront un impact sur *loadPreKey* et *loadPreKey* qui devront être adaptées pour prendre en compte ce nouveau paramètre booléen et cette fusion. Le *SQLiteAxolotlStore* devra aussi être adapté.

L'ajout de cette nouvelle colonne entrainera aussi des changements au niveau de l'insertion des clés. En effet, les méthodes *storePreKey* et *storeSignedPreKey* devront aussi être adaptées.

Les méthodes *loadSignedPreKeys* et *getSignedPreKeysCount* devront aussi être adaptées.

Dans le *SQLiteAxolotlStore*, *SIGNED_PREKEY_TABLENAME* peut alors être supprimée.

### 3.6 Scenario 6 : update de *fast_token* de la table *accounts* :

La mise à jour de la variable fast_token est prise en compte dans la classe Account. Il est possible de mettre à jour la valeur de celle-ci à null lors de l'invocation de la méthode *resetFastToken()* ou bien de mettre à jour la valeur selon un string entré en paramètre avec la méthode *setFastToken()*. Par la suite, il faut utiliser le DatabaseBackend pour mettre à jour l'Account via la méthode updateAccount.

### 3.7 Scenario 7 : split *accounts* en deux nouvelles tables

La table *accounts* est assez grande, environ 20 colonnes. Certaines colonnes stockent des informations propres à la connexion de l'utilisateur tandis que d'autres stockent des informations de profils des utilisateurs. Il pourrait dès lors être intéressant de scinder la table *accounts* en deux nouvelles tables : *acccounts_user_info* et *accounts_connection_info*. Les changements à mettre en place s'effectueraient surtout dans le fichier *DatabaseBackend.java* dans lequel il faudra retirer le code de création de la table *accounts* et rajouter la création des deux nouvelles tables. Comme dans les autres cas, une migration et un changement de version de la base de données devra être effectué. Une fois la base de données créée, deux choix sont possibles :

1. Propager les modifications dans les entités du code
2. Garder ces modifications uniquement au sein de la base de données.

Dans le premier cas, il faudra créer de nouvelles classes dans le dossier entité pour les nouvelles tables, supprimer l'ancienne entité *Account* et mettre à jour les bouts de code utilisant l'ancienne classe avec les nouvelles méthodes à appeler. Des requêtes plus avancées telles que des opérations de jointure devront potentiellement être mises en place pour pouvoir récupérer des informations présentes dans les deux tables simultanément.

Si l'on choisit la deuxième option, il suffit de modifier la classe *Account* en modifiant le comportement des fonctions faisant des requêtes à la base de données pour qu'elle prenne en compte le changement dans les tables. Cette solution est moins couteuse, car peu de code doit être modifié. Cependant, d'un point de vue de la modifiabilité

et l'évolution logiciel, cette option parait moins souhaitable, car elle instaure une vraie différence entre les interfaces que le code manipule et la base de données réelle.

### 3.8 Scenario 8 : changement du type de la colonne *uuid* de la table *accounts*

La colonne *uuid* de la table *accounts* a actuellement le type *TEXT* et permet de stocker un UUID généré de manière aléatoire par le package *java.util.UUID*. Il aurait aussi à l'origine été possible de simplement utiliser un id de type *INTEGER* qui s'autoincrémente à chaque création de nouveau compte. Si l'on venait à vouloir remplacer les uuids par des id auto-incrémentaux, cela aurait beaucoup d'impact sur la base de données et peu sur le code. Il faudrait en effet mettre en place une migration qui non seulement met à jour les identifiants, mais aussi met à jour les clés étrangères situées dans les tables *conversations*, *contacts*, *signed_prekeys*, *sessions*, *prekeys* et *identities*. Au niveau du code, il suffira de changer le type de l'identifiant dans la classe *Account* du dossier *entities* et de mettre à jour les types dans les portions de code manipulant cet identifiant.

### 3.9 Scenario 9 : insérer la nouvelle table *servers* (cfr schéma logique)

Afin de pouvoir insérer une table *server* comme dans notre sous-schéma logique, il est impératif de suivre le pattern DAO. De ce fait, il faut insérer une classe *Server* dans un fichier *Server.java* et ajouter des méthodes dans la classe *DatabaseBackend* pour gérer les interactions avec la base de données SQLite.

Il faut ensuite ajouter les strings de la création de cette table aux différentes queries de création initiale de la database SQLite, créer ou modifier les méthodes s'occupant de l'élaboration de l'envoi des queries la concernant, et ajouter l'exécution de la query dans une nouvelle condition de la méthode *onUpgrade*, chargée de la migration de bases de données de versions précédentes, tout cela au sein du fichier *DatabaseBackend.java*.

### 3.10 Scenario 10 : Renommage de la table *contacts*

Changer le nom d'une table est assez simple dans l'architecture du projet. Il suffit de changer le nom de la table dans le fichier *Contact* dans le dossier *entities*. Il s'agit là du seul changement minimal nécessaire. Si, pour des raisons de facilité de compréhension, on souhaitait renommer la classe *Contact* du dossier entities, les IDEs actuels (ici Android Studio) permettent le renommage automatique du nom de la classe et de tous les endroits dans le code où celui-ci est utilisé.

# 4 Étape 4 : Commentaires sur le schéma actuel et améliorations proposées

## 4.1 Points positifs du schéma et du code le manipulant

Le programme manipule la base de données à l'aide d'un DAO. Cela permet de propager efficacement tout changement fait dans la base de données, car il suffit de modifier la classe représentant la table en question et potentiellement modifier certains appels de méthodes depuis des classes externes. Les modifications sont ainsi fortement localisées dans les fichiers des entités et le fichier DatabaseBackend.java.

Cette structure permet ainsi de centraliser dans le code, là où les requêtes SQL sont décrites. Toutes les autres classes du programme ne manipulent pas directement la base de données mais les classes *entities* et les méthodes qu'elles implémentent. Mettre en place une nouvelle technologie de base de données est de ce fait facilement réalisable sans devoir modifier l'intégralité du code source.

Concernant la comparaison du schéma sur plusieurs versions de l'application, il est intéressant de noter que les développeurs ont créé un système de migration avec des versions de schéma qui sont codées dans l'application. De plus, à chaque nouvelle version de l'application, un nouveau commit avec pour titre "version bump to ..." a été créé, ce qui rend la tâche d'associer une version de schéma à une version de l'application plus facile.

## 4.2 Inconvénients détectés

### 4.2.1 Manque de consistance dans les conventions de nommage

Dans un premier temps, nous avons remarqué certaines inconsistances dans les noms de certaines colonnes. En effet, dans la plupart des tables, il semblerait que le snake case soit utilisé. Cependant, dans certaines tables, nous pouvons remarquer que certaines colonnes utilisent du camel case. C'est notamment le cas des colonnes *contactUuid*, *accountUuid* et *contactJid* de *conversations* qui devraient être nommées *contact_uuid*, *account_uuid* et *contact_jid*. Le cas le plus flagrant se trouve dans la table *messages* dont certaines colonnes utilisent le snake case et d'autres utilisent le camel case.

Il serait donc pertinent de mettre en place des conventions de nommage dans un souci d'uniformisation.

### 4.2.2 Tables inutiles

Certaines tables sont utilisées au niveau de la base de données, mais ne semblent avoir aucune implication au niveau de l'interface de l'application. En effet, la table *presence_templates* semble avoir été créée pour une feature qui a été abandonnée ou qui n'a pas encore été implémentée. Elle est utilisée en écriture lorsque l'utilisateur change son message de statut afin de créer un template qui devrait pouvoir être récupéré par le suite. Cependant, nous avons remarqué que des lectures sont effectuées pour récupérer ces templates mais les résultats ne sont jamais affichés à l'utilisateur. Cela nous laisse penser que cette table n'a plus (ou pas encore) d'utilité et devrait potentiellement être supprimée pour réduire la taille et la complexité de la base de données.

### 4.2.3 Utilisation de mots clés réservés

En SQLite, le mot clé "KEY" est un mot clé réservé. Or, dans les tables *signed_prekeys* et *prekeys*, certaines colonnes sont nommées "key". Bien que cela ne pose pas de problème dans l'application parce qu'il n'y a pas de requêtes raw sur ces tables, il est toujours délicat d'utiliser des noms de tables ou colonnes qui font partie des mots clés du langage. Ce problème nous est apparu lorsque nous avons essayé de parser les requêtes SQLite que nous avons obtenues en analysant les logs de l'application.

#### 4.2.4 Requêtes delete identiques

Lors de la suppression de compte, nous avons remarqué dans les logs SQLite que la query de suppression est faite à plusieurs reprises en rafales sans aucune raison apparente. Il s'agit là sûrement d'une erreur dans le code source de l'application qui mène à l'exécution de 8 fois la même requête de suppression.

#### 4.2.5 Mise à jour des données mises en cache

L'utilisation simultanée d'une base de donnée SQLite et d'un système de cache dans le code rend toute analyse portant la pertinence d'une table ou colonne bien plus complexe.

La table *contacts* ne récupère les données de la base de donnée qu'au démarrage de l'application. Cette opération fut tellement cachée que nous pensions à la base qu'aucun select n'était fait à la table *contacts* et qu'un autre mécanisme ou une autre table remplaçait celle-ci, mettant en doute sa pertinence.

Un exemple frappant survient notamment lors du changement de nom d'utilisateur. Lorsqu'un utilisateur a effectué le changement sur l'interface, celui-ci n'est pas mis à jour dans la base de données immédiatement. Il faut soit quitter l'application, soit effectuer une autre opération ayant un mécanisme de stockage différent, tel l'enregistrement d'un nouveau statut, pour voir apparaitre les changements au sein de la base de donnée.

#### 4.2.6 Informations encapsulées

En analysant la base de données, nous avons remarqué que la colonne *attributes* de la table *conversations* permettait de stocker plusieurs attributs sous forme de JSON. Certains attributs stockés dans ce JSON pourraient pour autant bénéficier d'une vraie colonne dans la base de données plutôt que d'être stocké dans une colonne *attributes*. Par exemple, le fait qu'une conversation soit mise en favori (qui est stocké dans *attribute*) pourrait se voir sortit de cette colonne et mis dans une nouvelle colonne *pinned_on_top*. Sans exécuter l'application, il est difficile de savoir que cette propriété est stockée dans la colonne *attributes*.

#### 4.2.7 Colonnes toujours NULL

L'analyse dynamique du code a permis de détecter certaines colonnes qui, dans tous les cas testés, restaient évaluées à NULL. Toutefois, du code permettant de mettre à jour la colonne est présent au sein de l'application. Il serait souhaitable de pouvoir faire une analyse avec un développeur connaissant la base de données afin de pouvoir obtenir une explication claire au niveau de l'utilisation réelle de colonnes dans l'application afin de les supprimer et de se débarrasser de possible code mort. Voici quelques exemples de colonnes pour lesquelles la valeur est toujours NULL :

- *bodyLanguage* de la table *messages*
- *systemname* de la table *contacts* (supposé prendre le surnom d'un contact dans un cas particulier non reproduit).

#### 4.2.8 Settings nombreux et cachés

L'application comporte un grand nombre de paramètres de configuration permettant d'activer ou non l'utilisation de certaines colonnes dans les tables. Par défaut, certaines colonnes ne sont pas utilisées. Le principal problème est qu'il est difficile de trouver l'impact qu'à un paramètre de configuration sur la base de données, ce qui rend l'analyse dynamique difficile à mettre en place. Il est compliqué de tester les valeurs stockées dans une colonne sans savoir quel paramètre permet d'activer son utilisation.

#### 4.2.9 Manque de documentation et de commentaires

Un problème récurrent des bases de données est le manque de documentation de celle-ci. Le projet *Conversation* ne fourni aucune documentation aux développeurs pour pouvoir appréhender le code et comprendre le fonctionnement de la base de données. Il serait bon de mettre en place un fichier reprenant le rôle de chacune des colonnes dans la base de données. De plus, insérer des commentaires dans le code manipulant la base de données permettrait de mieux comprendre la logique d'utilisation de celle-ci.