UNamur

Faculté d'informatique

INFOM218 :Evolution de systèmes logiciels

Rétro-Ingénierie d'une base de donnée

Auteurs :
Rosny Anderson Fodja Kouanga
Noé Pozza
Ryad HARITTANE

9 janvier 2025

Introduction

Ce rapport documente l'analyse approfondie de la base de données de l'application Android open-source **Conversations**. En nous appuyant sur l'outil **SQLInspect**, nous avons étudié les interactions SQL présentes dans le code source pour reconstituer et analyser les schémas physiques et logiques de la base de données. Cette démarche vise à mieux comprendre la structure des données, leur utilisation dans l'application et les opportunités d'amélioration.

Le code source de **Conversations**, ainsi que les résultats détaillés de notre analyse, sont centralisés dans un dépôt GitHub dédié : `lien vers le repo`. Il constitue une ressource complète pour reproduire nos étapes, explorer le projet ou y contribuer.

Le rapport est structuré comme suit : Dans un premier temps, nous procédons à l'analyse du code DDL pour identifier et reconstruire les structures de la base de données de l'application. Ensuite, nous présentons le schéma physique obtenu, tout en explorant les relations implicites et explicites entre les différentes entités. Une section est dédiée aux propositions d'améliorations, incluant la découverte de clés étrangères implicites et l'ajout de contraintes pour renforcer l'intégrité des données. Nous examinons ensuite l'impact des scénarios d'évolution sur la structure de la base de données et sur le code source de l'application. Enfin, nous concluons en évaluant la qualité globale de la base de données et en formulant des recommandations pour sa maintenance et son évolution futures.

En suivant cette approche méthodique, nous visons à fournir une compréhension approfondie de la base de données utilisée dans **Conversations** et à proposer des solutions concrètes pour la rendre plus robuste, efficace et évolutive.

Analyse du code DDL

Dans cette section, nous allons commencer notre rétro-ingénierie de base de donnée par l'analyse du code DDL présent dans la code base de l'application. Ceci nous permettra de ressortir des informations tant bien sur l'aspect technique que sur la signification des données. Pour faciliter notre travail, on a aggloméré les informations dans un fichier *MarkDown* nommé *analyse.md*,disponible sur notre répertoire.

Classe d'intérêt

En premier lieu, il est nécessaire de trouver la, ou les, parties de l'application responsable de la gestion de la base de données. Le rapport fourni par SQLInspect nous dirige vers deux classes : *DatabaseBackend* et *UnifiedPushDatabase*. Le point commun de ces deux classes est qu'elles sont filles de la classe abstraite *SQLiteOpenHelper*. Cette classe permet la gestion de bases de données SQL.

Sachant que le moteur de base de données utilisé est SQLite, ce qui est courant

pour une application mobile, nous constatons qu'il existe deux bases de données distinctes. Cette conclusion découle de l'existence de deux classes distinctes implémentant la classe *SQLiteOpenHelper*, chacune étant responsable de la gestion d'une base de données spécifique.

DatabaseBackend

La classe *DatabaseBackend* définit la base de données nommée *History*. Celle-ci semble avoir la responsabilité de contenir les données relatives aux messages, à la sécurité, aux contacts et aux serveurs. Elle est donc la base de données principale de cette application Android.
Elle contient 11 tables différentes. Nous avons reconstitué les requêtes DDL à l'origine des classes afin de pouvoir recréer le schéma physique. Voici la liste :

Contact

Contient les informations sur un contact. Elle fait partie de la fonctionnalité de gestion des contacts. Il y est définie une relation avec la table *Account*. L'application permettant de gérer plusieurs comptes, il est nécessaire de lier un contact à un compte.

Un contact peut être aussi un groupe, ou plus formellement, un *Multi-User Chat (MUC)*. C'est une fonctionnalité du protocole XMPP, qui permet de créer des salons de discussion.

D'après nos analyses , cette table ne possède pas de clé primaire explicite, mais des contraintes d'unicité sont appliquées sur les colonnes *accountUuid* et *jid*. Cela vise à empêcher l'enregistrement de doublons pour un même contact. Cependant, il apparaît que ces colonnes, *jid* et *accountUuid*, sont utilisées implicitement comme une clé primaire [1]. Cela peut poser un problème, car la valeur *null* est permise pour ces colonnes, ce qui pourrait nuire à l'intégrité des données.

Le champ *jid* (mis pour *Jabber ID*) est un identifiant unique utilisé dans le protocole XMPP, un protocole de messagerie instantanée sur lequel repose l'application.

De plus, il est intéressant de noter que cette table est l'une des rares à ne pas hériter de la classe *AbstractEntity*, ce qui se traduit par l'absence de la colonne *uuid*, habituellement utilisée comme identifiant unique dans les autres entités.

DDL

```
create table contacts(
    accountUuid TEXT,
    servername TEXT,
    systemname TEXT,
    presence_name TEXT,
```

1. eu.siacs.conversations.entities.Contact 115 : *Jid.of* n'accepte pas de valeur null

```
    jid TEXT,
    pgpkey TEXT,
    photouri TEXT,
    options NUMBER,
    systemaccount NUMBER,
    avatar TEXT,
    last_presence TEXT,
    last_time NUMBER,
    rtpCapability TEXT,
    groups TEXT,

    FOREIGN KEY(accountUuid) REFERENCES accounts(uuid)
     ON DELETE CASCADE,
    UNIQUE(accountUuid, jid) ON CONFLICT REPLACE
);
```

Account

Contient les informations sur un compte. Table centrale de cette base de données, elle permet la gestion des comptes XMPP. Il est donc à noter que les colonnes *username*, *server* et *ressource* font donc référence au *jid* utilisé par le protocole XMPP. Il prend la forme de *username@server/ressource*[2].

Tout comme pour la table *Contact*, la colonne *jid* semble ne pas pouvoir être *null*. Une requête exécutée avec une valeur *null* pour les colonnes *username* ou *server* entraîne une erreur[3]. Cela confirme implicitement que *jid* doit être bien formé et non *null* pour garantir le fonctionnement correct de l'application.

Cette table possède cependant une clé primaire explicite, *uuid*, qui est une chaîne de caractères générée aléatoirement.

DDL

```
create table accounts(
    uuid TEXT PRIMARY KEY,
    username TEXT,
    server TEXT,
    password TEXT,
    display_name TEXT,
    status TEXT,
    status_message TEXT,
    rosterversion TEXT,
    options NUMBER,
    avatar TEXT,
    keys TEXT,
    hostname TEXT,
```

2. xmpp documentation
3. eu.siacs.conversations.entities.Account 176-180 : *Jid.of* n'accepte pas de valeur *null*

```
    resource TEXT,
    pinned_mechanism TEXT,
    pinned_channel_binding TEXT,
    fast_mechanism TEXT,
    fast_token TEXT,
    port NUMBER DEFAULT 5222
);
```

Conversation

Cette table contient les informations sur une conversation entre un compte et un contact. Elle est donc une table de liaison entre les tables *Account* et *Contact*. Elle contient aussi des informations sur le nom de la conversation, le statut, le mode et des attributs.

Deux colonnes se reportent implicitement à la table *Contact*. Il s'agit de *contactUuid* et *contactJid*. La première colonne est étrange , compte tenu du fait que la table *Contact* ne contient pas de colonne *uuid*. Il est possible que cette colonne soit une erreur de conception. La seconde colonne, *contactJid*, est une référence au *jid* du contact (qui peut donc être un groupe). Ce qui tend à confirmer que le *jid* est une clé primaire implicite de la table *Contact*[4].

Fille de la classe *AbstractEntity*, elle contient une clé primaire , *uuid*, qui est une chaîne de caractères générée aléatoirement.

DDL

```
create table conversations (
    uuid TEXT PRIMARY KEY,
    name TEXT,
    contactUuid TEXT,
    accountUuid TEXT,
    contactJid TEXT,
    created NUMBER,
    status NUMBER,
    mode NUMBER,
    attributes TEXT,

    FOREIGN KEY(accountUuid) REFERENCES accounts(uuid)
     ON DELETE CASCADE
);
```

4. eu.siacs.conversations.entities.Conversation 673 : Seule la colonne *contactJid* est utilisé comme lien avec la table *Contact*

Message

Contient les informations sur un message. Il est à noter que la colonne *conversationUuid* est une clé étrangère vers la table *Conversation*. Cette table contient aussi des informations sur le contenu du message et sur l'aspect technique du message, comme le type, le statut, le temps d'envoi, etc.

La colonne *trueCounterpart* est une colonne qui sert à stocker le *jid* du contact, lorsque la conversation est un groupe. Cela permet de savoir qui a envoyé le message. Elle est donc en un sens une clé étrangère implicite vers la table *Contact*[5].

Une table virtuelle est aussi présente. Sous le nom de *messages_index*, elle permet une recherche rapide des messages. Elle est donc une table de cache. Afin de maintenir à jour cette table, des triggers sont utilisés afin que chaque modification, ajout, suppression de la table *Message* soit répercutée sur la table *messages_index*.

DDL

```
create table messages(
    uuid TEXT PRIMARY KEY,
    conversationUuid TEXT,
    timeSent NUMBER,
    counterpart TEXT,
    trueCounterpart TEXT,
    body TEXT,
    encryption NUMBER,
    status NUMBER,
    type NUMBER,
    relativeFilePath TEXT,
    serverMsgId TEXT,
    axolotl_fingerprint TEXT,
    carbon INTEGER,
    edited TEXT,
    read NUMBER DEFAULT 1,
    oob INTEGER,
    errorMsg TEXT,
    readByMarkers TEXT,
    markable NUMBER DEFAULT 0,
    deleted NUMBER DEFAULT 0,
    bodyLanguage TEXT,
    occupantId TEXT,
    reactions TEXT,
    remoteMsgId TEXT,

    FOREIGN KEY(conversationUuid) REFERENCES conversations(uuid)
     ON DELETE CASCADE
);
```

5. eu.siacs.conversations.entities.Message 352 : *trueCounterpart* est utilisé pour la relation avec *Contact*

Identity

Table servant à stocker les informations sur les identités. Il s'agit des informations sur les clés cryptographiques. Elle contient des informations sur le compte, le nom, la clé, le certificat, la confiance, etc.

Étant liée à l'aspect sécurité de l'application, nous n'allons pas nous attarder sur les détails de cette table.

DDL

```
CREATE TABLE identities(
    account TEXT,
    name TEXT,
    ownkey INTEGER,
    fingerprint TEXT,
    certificate BLOB,
    trust TEXT,
    active NUMBER,
    last_activation NUMBER,
    key TEXT,

    FOREIGN KEY(account) REFERENCES accounts(uuid)
     ON DELETE CASCADE
    UNIQUE( account, name, fingerprint)
     ON CONFLICT IGNORE
);
```

Session

Table servant à stocker les informations sur les sessions. Elle contient des informations sur le compte, le nom, l'identifiant du device, la clé, etc.

Étant liée à l'aspect sécurité de l'application, nous n'allons pas nous attarder sur les détails de cette table.

DDL

```
CREATE TABLE sessions(
    account TEXT,
    name TEXT,
    device_id INTEGER,
    key TEXT,

    FOREIGN KEY(account) REFERENCES accounts(uuid)
     ON DELETE CASCADE,
    UNIQUE( account, name, device_id)
     ON CONFLICT REPLACE
```

```
);
```

Prekey

Étant liée à l'aspect sécurité de l'application, nous n'allons pas nous attarder sur les détails de cette table.

DDL

```
CREATE TABLE prekeys(
    account TEXT,
    id INTEGER,
    key TEXT,
    FOREIGN KEY(account) REFERENCES accounts(uuid)
     ON DELETE CASCADE,
    UNIQUE( account, id)
     ON CONFLICT REPLACE
);
```

Signed Prekey

Étant liée à l'aspect sécurité de l'application, nous n'allons pas nous attarder sur les détails de cette table.

DDL

```
CREATE TABLE signed_prekeys(
    account TEXT,
    id INTEGER,
    key TEXT,

    FOREIGN KEY(account) REFERENCES accounts(uuid)
     ON DELETE CASCADE,
    UNIQUE( account, id)
     ON CONFLICT REPLACE
);
```

Resolver Result

Cette table contient les résultats de la résolution DNS. Elle contient des informations sur le domaine, le nom d'hôte, l'adresse IP, la priorité, le TLS direct, l'authentification, le port, etc.

Elle permet à l'application de garder en mémoire les résultats de résolution DNS pour une utilisation ultérieure.

DDL

```
create table resolver_results(
    domain TEXT,
    hostname TEXT,
    ip BLOB,
    priority NUMBER,
    directTls NUMBER,
    authenticated NUMBER,
    port NUMBER,

    UNIQUE(domain) ON CONFLICT REPLACE
);
```

Discovery Result

Cette table contient le résultat des découvertes de services disponibles . Elle est liée aux fonctionnalités de XMPP [6].

DDL

```
create table discovery_results(
    hash TEXT,
    ver TEXT,
    result TEXT,

    UNIQUE(hash, ver) ON CONFLICT REPLACE
);
```

Presence Template

Contient les informations sur les patterns de présence. C'est-à-dire, un message que l'utilisateur veut afficher, et un statut. Ce statut peut être *chat*, *online*, *away*, *offline*, etc [7].

On constate qu'aucune des colonnes n'est une clé primaire. Cependant, une contrainte d'unicité est appliquée sur les colonnes *message* et *status*. Cela permet de ne pas avoir deux fois le même message avec le même statut.

DDL

```
CREATE TABLE presence_templates(
    uuid TEXT,
    last_used NUMBER,
    message TEXT,
    status TEXT,
```

6. XMPP Documentation
7. eu.siacs.conversations.entities.Presence 12

```
    UNIQUE(message, status) ON CONFLICT REPLACE
);
```

UnifiedPushDatabase

Cette classe définit la base de données nommée *unified-push-distributor*. Celle-ci semble avoir la responsabilité de contenir les données relatives aux notifications push. Elle ne contient qu'une seule table, *push*, qui contient les informations sur les notifications push.

Push

Contient les informations sur une notification push. Elle est reliée au système de notification d'Android .

Elle semble liée à la table *Account*, mais il n'y a pas de clé étrangère déclarée. Cependant, la colonne *account* est utilisée pour se référer à *accounts.uuid*, suggérant une clé étrangère implicite. L'utilisation de *account* comme clé étrangère implicite est sans doute contrainte par le fait que cette table et la table *Account* sont dans des bases de données différentes.

DDL

```
CREATE TABLE push (
        account TEXT,
        transport TEXT,
        application TEXT NOT NULL,
        instance TEXT NOT NULL UNIQUE,
        endpoint TEXT,
        expiration NUMBER DEFAULT 0
);
```

Analyse du Schéma et des Requêtes

Cette section détaille l'analyse des requêtes utilisées dans l'application. Elle met en évidence la découverte de clés étrangères implicites non représentées explicitement dans le schéma physique obtenu. Nous examinons également dans quelle mesure les clés étrangères explicites peuvent être inférées à partir du schéma physique ou des requêtes utilisées dans l'application. De plus, cette section explore la possibilité de découvrir des clés étrangères implicites qui ne sont pas explicitement déclarées. L'objectif est de compléter le schéma logique en identifiant et en ajoutant ces relations non documentées, afin de renforcer l'intégrité référentielle et la clarté du modèle de données.

Résultats

Inférence des Clés Étrangères Explicites

Pour ce faire, nous proposons de procéder en deux étapes : tout d'abord, en examinant le schéma physique (en se basant sur les noms et types des colonnes) pour identifier d'éventuelles clés étrangères. Ensuite, en analysant les requêtes SQL, notamment les jointures, les relations entre sorties et entrées des requêtes, afin de confirmer ou de découvrir des liens implicites supplémentaires entre les tables.

En analysant le schéma physique, on observe qu'une grande majorité des tables contiennent une colonne `account` de type `TEXT`. Par déduction logique, et en nous appuyant sur les résultats fournis par SQLInspect ainsi que sur les jointures détectées, nous pouvons supposer que ces colonnes `account` font référence à la colonne `Account.uuid`. Cette relation semble être une clé étrangère implicite, bien qu'elle ne soit pas explicitement définie dans le schéma physique.

De plus, plusieurs relations implicites entre les tables peuvent être identifiées :

La table `Account` contient une colonne **hostname**, que l'on peut relier logiquement à la colonne **hostname** de la table `Resolver_Results`. Cette liaison pourrait permettre de retrouver les résultats de résolutions DNS associés à un compte donné.

La table `Conversations` possède une colonne **ContactJid**, qui semble être liée à la colonne **ContactJid** de la table `Contacts`. Cette relation établit une correspondance entre une conversation et le contact auquel elle est associée.

La table `Message` comprend une colonne **ConversationUuid**, que l'on relie à la colonne **uuid** de la table `Conversations`. Cette relation lie chaque message à une conversation spécifique, garantissant ainsi la cohérence des données entre ces deux entités.

En ce qui concerne les requêtes, elles ne fournissent aucune information supplémentaire, ni sur les jointures ni sur d'autres relations. Sans une analyse plus approfondie du programme, il n'est pas possible d'en tirer des conclusions supplémentaires.

Calculons maintenant le recall et la précision comme cela est fait dans le cours. Pour ce qui est de la précision c'est simple, c'est **100%**, en effet toutes les clés étrangères sont des vraies et aucune n'est un faux positif.

Par contre pour le recall étant donné que l'on n'a pas trouvé toutes les clés, il faut faire les calculs par rapport aux nombres de clés totales. On a déduits du schéma **10 clés étrangères** pour un total de **14 clés étrangères**. Notre recall est donc évalué à **71,42%**.

Découverte de Clés Étrangères Implicites Additionnelles

Nous avons analysé les requêtes SQL pour identifier des utilisations de colonnes qui fonctionnent comme des clés étrangères mais qui ne sont pas formellement définies comme

telles dans les déclarations de table.

- **Relation** `push → accounts` **:**
  - **Requête :** `SELECT EXISTS(SELECT endpoint FROM push WHERE account = ...)`
  - **Construction :** `push.account` → `accounts.uuid`
  - **Justification :** Cette requête utilise `push.account` pour se référer à `accounts.uuid`, suggérant une clé étrangère entre ces deux tables.
- **Relation** `conversations.contactJid → contacts.jid` **:**
  - **Requête :** Utilisation de `contactJid` dans la classe `Conversation.java` (ligne 673).
  - **Construction :** `conversations.contactJid` → `contacts.jid`
  - **Justification :** `contactJid` est utilisé comme lien avec `contacts.jid` dans la logique de l'application. Seule la colonne `contactJid` est utilisée pour établir cette relation, tandis que `contactUuid` reste inutilisée, ce qui souligne une dépendance implicite.
- **Relation** `messages → contacts` **:**
  - **Requête :** Références dans `Message.java` (lignes 344-355).
  - **Construction :** `messages.trueCounterpart` → `contacts.jid`
  - **Justification :** `trueCounterpart` est utilisé pour référencer `contacts.jid`, dépendant du mode de conversation (un-à-un ou groupe). Si la conversation est entre deux personnes, `trueCounterpart` fait directement référence à `null`. Sinon, dans un contexte de groupe, `trueCounterpart` recherche l'identité appropriée dans la liste des contacts.
- **Relation** `messages → contacts` **:**
  - **Construction :** `messages.counterpart` → `contacts.jid`
  - **Justification :** `counterpart` est utilisé pour définir le destinataire principal d'un message en fonction du type de conversation défini dans `Conversations.mode` (individuelle, privée ou de groupe).
    - Dans une conversation individuelle (`Conversation.MODE_SINGLE`), `counterpart` fait directement référence au `jid` du contact avec lequel l'utilisateur échange. Ce JID est utilisé pour définir la destination du paquet XMPP via la méthode `packet.setTo()`.
    - Dans une conversation privée au sein d'un groupe (`message.isPrivateMessage()`), `counterpart` fait référence au JID complet (`FullJid`) du membre du groupe recevant le message privé.
    - Dans une conversation de groupe standard (`Type.GROUPCHAT`), `counterpart` correspond au `jid` du groupe (`asBareJid()`), ce qui définit l'ensemble du groupe comme destinataire du message.
- **Relation** `sessions → contacts ou Account` **:**
  - **Requête :** Références dans `AxolotlService.java`.
  - **Construction :** `sessions.name` → `contacts.jid` ou `accounts.jid`
  - **Justification :** `name` dans la table `sessions` est utilisé pour identifier l'origine d'une session OMEMO, en fonction du type de JID (contact ou compte local).
    - Lorsque la session concerne un contact, `name` est initialisé à partir de `contact.getJid()` via la méthode `findSessionsForContact()`. Cela correspond au JID complet du contact.
    - Lorsque la session concerne l'utilisateur local, `name` est défini via `account.getJid().asBareJid().toString()` dans `getOwnAxolotlAddress()`. Cela correspond au JID sans

ressource (bare JID) de l'utilisateur. et des méthodes comme `buildSessionFromPEP(final SignalProtocolAddress address, OnSessionBuildFromPep callback)` montre qu'une session peut s'ouvrir avec son propre compte sur un autre device (différents `device_id`).
    - Cette dualité permet de stocker dans une même structure (la table `sessions`) des sessions pour les contacts ainsi que les sessions propres à l'utilisateur local, tout en distinguant ces deux cas selon le contexte.
- **Relation `identities → accounts et contacts` :**
    - **Construction :** `identities.name` → `accountsjid (= Account.name@Account.server)` et `contacts.jid`, `identities.account` → `accounts.uuid`.
    - **Justification :**
        - Dans le cas d'un `Account JID`, la requête utilise `name` pour rechercher les clés associées à l'utilisateur local via son `accounts.jid` (exemple : `loadOwnIdentityKeyPair` dans `DatabaseBackend.java`). De plus, `identities.account` relie cette identité à un compte spécifique via `accounts.uuid`.
        - Dans le cas d'un `Contact JID`, `name` fait référence à `contacts.jid`, comme illustré dans la méthode `loadIdentityKeys`, qui récupère les clés d'un contact donné.
        - Cette dualité démontre que la table `identities` est liée à la fois à `accounts` et `contacts` en fonction du contexte d'utilisation.

La présence de clés étrangères non déclarées, comme celle entre `push` et `accounts`, indique que le schéma physique pourrait être amélioré pour expliciter ces relations, ce qui renforcerait la compréhension et la gestion de la base de données. L'ajout de cette clé au schéma logique aidera à maintenir l'intégrité référentielle et à optimiser les performances des requêtes.

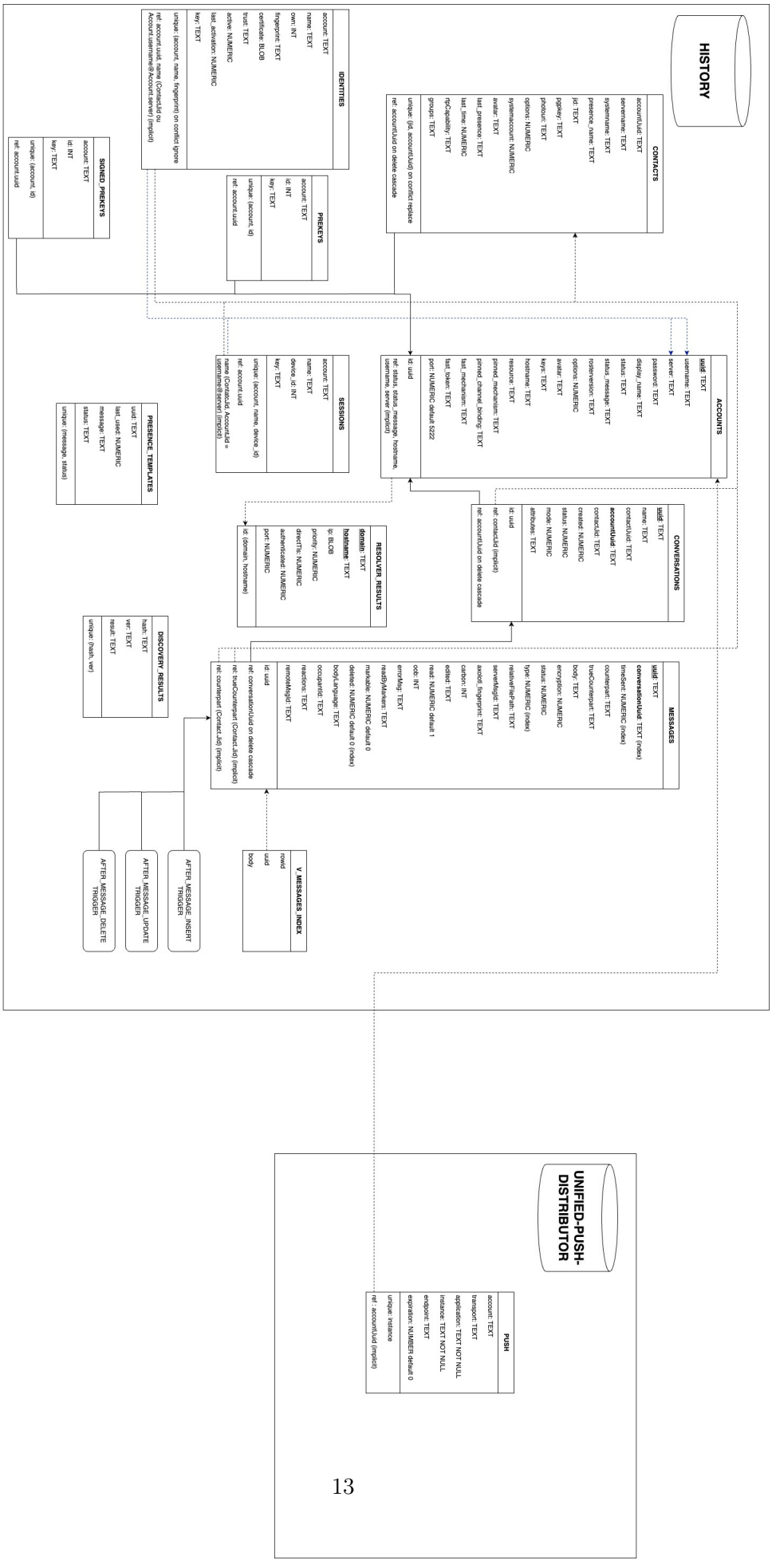
Brève description du Schéma Physique (PS)

Dans cette section, il s'agira de donner une brève description du schéma physique se trouvant sur notre repository Git dans le fichier /analyse/schema/schema_physique.drawio. Les autres schémas se trouvant aussi sur notre repository .

La principale différence observable par rapport à un schéma classique réside dans la représentation des liaisons de clés étrangères. Dans ce cas, les clés étrangères implicites sont représentées par des traits en pointillés noirs, tandis que les traits en pointillés bleus indiquent des clés étrangères implicites référencées sur deux colonnes d'une même table. Par exemple, la colonne `Identities.name` référence simultanément `Account.username` et `Account.server`.

La table `message_index` est une table virtuelle conçue pour faciliter les recherches rapides de messages. Elle agit comme un index spécialisé, optimisant les performances lors de requêtes complexes ou fréquentes sur les messages.

Il y a deux bases de données, l'une contenant toute l'information `History` et l'autre servant pour les notifications seulement `Unified-Push-Distributor`.


HISTORY
CONTACTS
accountUuid: TEXT
serverName: TEXT
systemname: TEXT
presence_name: TEXT
jid: TEXT
pgpKey: TEXT
photoId: TEXT
options: NUMERIC
systemaccount: NUMERIC
avatar: TEXT
last_presence: TEXT
last_time: NUMERIC
ntpCapability: TEXT
groups: TEXT
unique: (jid, accountUuid) on conflict replace
ref: accountUuid on delete cascade
IDENTITIES
account: TEXT
name: TEXT
own: INT
fingerprint: TEXT
certificate: BLOB
trust: TEXT
active: NUMERIC
last_activation: NUMERIC
key: TEXT
unique: (account, name, fingerprint) on conflict ignore
ref: account.uuid, name (ContactId.ou Account.username@Account.server) (implicit)
PREKEYS
account: TEXT
id: INT
key: TEXT
unique: (account, id)
ref: account.uuid
SIGNED_PREKEYS
account: TEXT
id: INT
key: TEXT
unique: (account, id)
ref: account.uuid
ACCOUNTS
uuid: TEXT
username: TEXT
server: TEXT
password: TEXT
display_name: TEXT
status: TEXT
status_message: TEXT
rosterversion: TEXT
options: NUMERIC
avatar: TEXT
keys: TEXT
hostname: TEXT
resource: TEXT
pinned_mechanism: TEXT
pinned_channel_binding: TEXT
fast_mechanism: TEXT
fast_token: TEXT
port: NUMERIC default 5222
id: uuid
ref: status, status_message, hostname, username, server (implicit)
SESSIONS
account: TEXT
name: TEXT
device_id: INT
key: TEXT
unique: (account, name, device_id)
ref: account.uuid
name (ContactJid, AccountJid = username@server) (implicit)
CONVERSATIONS
uuid: TEXT
name: TEXT
contactJuid: TEXT
accountUuid: TEXT
contactJid: TEXT
created: NUMERIC
status: NUMERIC
mode: NUMERIC
attributes: TEXT
id: uuid
ref: contactJid (implicit)
ref: accountUuid on delete cascade
MESSAGES
uuid: TEXT
conversationUuid: TEXT (index)
timeSent: NUMERIC (index)
counterpart: TEXT
trueCounterpart: TEXT
body: TEXT
encryption: NUMERIC
status: NUMERIC
type: NUMERIC (index)
relativeFilePath: TEXT
serverMsgId: TEXT
axolotl_fingerprint: TEXT
carbon: INT
edited: TEXT
read: NUMERIC default 1
oob: INT
errorMsg: TEXT
readByMarkers: TEXT
markable: NUMERIC default 0
deleted: NUMERIC default 0 (index)
bodyLanguage: TEXT
occupantId: TEXT
reactions: TEXT
remoteMsgId: TEXT
id: uuid
ref: conversationUuid on delete cascade
ref: trueCounterpart (ContactJid) (implicit)
RESOLVER_RESULTS
domain: TEXT
hostname: TEXT
ip: BLOB
priority: NUMERIC
directTls: NUMERIC
authenticated: NUMERIC
port: NUMERIC
id: (domain, hostname)
PRESENCE_TEMPLATES
uuid: TEXT
last_used: NUMERIC
message: TEXT
status: TEXT
unique: (message, status)
DISCOVERY_RESULTS
hash: TEXT
ver: TEXT
result: TEXT
unique: (hash, ver)
V_MESSAGES_INDEX
rowid
uuid
body
AFTER_MESSAGE_INSERT TRIGGER
AFTER MESSAGE_UPDATE TRIGGER
AFTER MESSAGE_DELETE TRIGGER
UNIFIED-PUSH-DISTRIBUTOR
PUSH
account: TEXT
transport: TEXT
application: TEXT NOT NULL
instance: TEXT NOT NULL
endpoint: TEXT
expiration: NUMBER default 0
unique: instance
ref : accountUuid (implicit)

Statistics about the database queries

Statistiques

Métriques grâce à SQLInspect

Grâce à l'outil `SQLInspect`, nous avons pu collecter les métriques suivantes sur les interactions SQL dans le projet, réparties par fichier :

- **ImportBackupService.java** :
  - `Queries` : 1
  - `Inserts` : 0
  - `Selects` : 1
  - `Updates` : 0
  - `Deletes` : 0
- **DatabaseBackend.java** :
  - `Queries` : 112
  - `Inserts` : 1
  - `Selects` : 10
  - `Updates` : 8
  - `Deletes` : 3
- **UnifiedPushDatabase.java** :
  - `Queries` : 2
  - `Inserts` : 0
  - `Selects` : 1
  - `Updates` : 0
  - `Deletes` : 0
- **ExportBackupWorker.java** :
  - `Queries` : 1
  - `Inserts` : 0
  - `Selects` : 1
  - `Updates` : 0
  - `Deletes` : 0

Limitations de SQLInspect

Bien que l'outil `SQLInspect` ait permis d'identifier une grande partie des interactions SQL dans le projet, il présente certaines limitations importantes. Notamment, `SQLInspect` n'est pas en mesure de reconstruire les requêtes SQL formées dynamiquement en Java à l'aide des méthodes `.query()`, `.insert()`, `.delete()` et `.update()`. Cela entraîne un sous-dénombrement des interactions SQL dans les fichiers analysés.

En analysant manuellement le code, nous avons pu identifier les requêtes SQL supplémentaires suivantes :

- **DatabaseBackend.java** :

  - `Insert` : 12 requêtes supplémentaires.
  - `Select` : 21 requêtes supplémentaires.
  - `Delete` : 15 requêtes supplémentaires.
  - `Update` : 16 requêtes supplémentaires.
- **UnifiedPushDatabase.java** :
  - `Insert` : 1 requête supplémentaire.
  - `Select` : 6 requêtes supplémentaires.
  - `Delete` : 3 requêtes supplémentaires.
  - `Update` : 1 requête supplémentaire.
- **ImportBackupService.java** :
  - `Insert` : 1 requête supplémentaire.
- **ExportBackupWorker.java** :
  - `Select` : 2 requêtes supplémentaires.

Impact des limitations : Ces requêtes manquantes montrent que les résultats fournis par `SQLInspect` sous-estiment l'activité SQL réelle dans le projet. Cela met en évidence la nécessité de combiner les analyses automatiques avec une validation manuelle pour obtenir une image complète des interactions SQL.

Ces métriques fournissent une vue d'ensemble de l'activité SQL à travers les différents fichiers du projet et mettent en lumière les zones critiques, comme **DatabaseBackend.java**, qui contient la majorité des requêtes SQL.

Rôles des fichiers

Chaque fichier analysé dans ce projet a un rôle spécifique en lien avec les interactions SQL et la gestion des données. Voici une description détaillée des responsabilités de chaque fichier :

- **ImportBackupService.java** :
  - Ce fichier est responsable de l'importation des données, notamment les informations liées à un compte utilisateur.
  - Les requêtes SQL ici servent principalement à insérer ou récupérer des données importées pour les intégrer dans les tables appropriées.
- **ExportBackupWorker.java** :
  - Ce fichier gère l'exportation des informations d'un compte utilisateur.
  - Les interactions SQL consistent principalement à extraire les données nécessaires pour créer un fichier de sauvegarde ou transférer les informations.
- **UnifiedPushDatabase.java** :
  - Ce fichier est chargé de la gestion des notifications push.
  - Les requêtes SQL ici concernent principalement la gestion des endpoints, des tokens d'authentification et des informations liées aux notifications.
- **DatabaseBackend.java** :
  - Ce fichier contient la logique principale pour toutes les interactions SQL du projet.
  - Il centralise les opérations complexes telles que l'insertion, la mise à jour, la suppression et la récupération des données.

— La majorité des requêtes SQL identifiées se trouvent dans ce fichier, ce qui en fait le cœur de la gestion des données dans l'application.

Conclusion : Chaque fichier joue un rôle précis et complémentaire dans le projet. Cependant, la logique principale et la majorité des interactions SQL résident dans **DatabaseBackend.java**, qui sert de base à toutes les fonctionnalités de gestion des données.

Analyse de complexité des requêtes SQL

Dans le fichier `DatabaseBackend.java`, les requêtes SQL identifiées présentent une complexité variable selon leur type et leur usage. Dans la plupart des cas, elles restent simples et intuitives. Voici une analyse détaillée, accompagnée d'exemples concrets tirés du code.

— **INSERT** :
  — La majorité des requêtes `INSERT` insèrent simplement des données sans complexité particulière. Exemple :

```
public void createMessage(Message message) {
    SQLiteDatabase db = this.getWritableDatabase();
    db.insert(Message.TABLENAME, null, message.getContentValues());
}
```

  — Quelques cas incluent une étape préalable, comme un `DELETE`, augmentant légèrement la complexité. Exemple :

```
public void insertPresenceTemplate(PresenceTemplate template) {
    SQLiteDatabase db = this.getWritableDatabase();
    db.delete(PresenceTemplate.TABELNAME,
              PresenceTemplate.UUID + " not in (select " +
              PresenceTemplate.UUID + " from " +
              PresenceTemplate.TABELNAME + " order by " +
              PresenceTemplate.LAST_USED + " desc limit 9)", null);
    db.insert(PresenceTemplate.TABELNAME, null,
    template.getContentValues());
}
```

— **SELECT** :
  — La plupart des `SELECT` récupèrent des données directement, sans jointures ni filtres complexes. Exemple :

```
public List<PresenceTemplate> getPresenceTemplates() {
    ArrayList<PresenceTemplate> templates = new ArrayList<>();
    SQLiteDatabase db = this.getReadableDatabase();
    Cursor cursor = db.query(PresenceTemplate.TABELNAME,
                             null, null, null, null, null,
                             PresenceTemplate.LAST_USED + " desc");
    while (cursor.moveToNext()) {
        templates.add(PresenceTemplate.fromCursor(cursor));
    }
```

```
        cursor.close();
        return templates;
    }
```

— Certains cas incluent des jointures complexes et des filtres avancés. Exemple :

```
    public Cursor getMessageSearchCursor(final List<String> term,
    final String uuid) {
        final SQLiteDatabase db = this.getReadableDatabase();
        final StringBuilder SQL = new StringBuilder();
        SQL.append("SELECT messages.*, conversations.contactJid, ")
            .append("conversations.account, conversations.mode ")
            .append("FROM messages JOIN conversations ON ")
            .append("messages.conversation = conversations.uuid ")
            .append("WHERE messages.body MATCH ?");
        if (uuid != null) {
            SQL.append(" AND conversations.uuid = ?");
        }
        return db.rawQuery(SQL.toString(), selectionArgs);
    }
```

— **UPDATE** :

— Les requêtes `UPDATE` sont souvent conditionnelles, mais restent simples. Exemple :

```
    public boolean updateMessage(final Message message,
    final boolean includeBody) {
        final var db = this.getWritableDatabase();
        final var contentValues = message.getContentValues();
        if (!includeBody) {
            contentValues.remove(Message.BODY);
        }
        return db.update(Message.TABLENAME, contentValues,
                         Message.UUID + "=?",
                         new String[]{message.getUuid()}) == 1;
    }
```

— Certains cas incluent des transactions, augmentant la complexité. Exemple :

```
    public void markFilesAsChanged(List<FilePathInfo> files) {
        SQLiteDatabase db = this.getReadableDatabase();
        db.beginTransaction();
        for (FilePathInfo info : files) {
            ContentValues contentValues = new ContentValues();
            contentValues.put(Message.DELETED, info.deleted ? 1 : 0);
            db.update(Message.TABLENAME, contentValues,
                      Message.UUID + "=?",
                      new String[]{info.uuid.toString()});
        }
        db.setTransactionSuccessful();
        db.endTransaction();
    }
```

— **DELETE** :

— Les `DELETE` sont majoritairement simples, basées sur des conditions uniques. Exemple :

```
public boolean deleteAccount(final Account account) {
    final var db = this.getWritableDatabase();
    return db.delete(Account.TABLENAME,
                     Account.UUID + "=?",
                     new String[]{account.getUuid()}) == 1;
}
```

— Quelques cas incluent des conditions multiples. Exemple :

```
public void deleteAllSessions(Account account,
SignalProtocolAddress contact) {
    SQLiteDatabase db = this.getWritableDatabase();
    String[] args = {account.getUuid(), contact.getName()};
    db.delete(SQLiteAxolotlStore.SESSION_TABLENAME,
              SQLiteAxolotlStore.ACCOUNT + "=? AND " +
              SQLiteAxolotlStore.NAME + " = ?", args);
}
```

Conclusion : Dans l'ensemble, les requêtes SQL du fichier `DatabaseBackend.java` sont **simples et intuitives** à comprendre. Les scénarios complexes, tels que les transactions ou les jointures, restent cohérents et logiques.

`SQLInspect` offre une analyse utile de la complexité, notamment en identifiant les imbrications dans les requêtes. Cependant, il ne capture pas la majorité des requêtes dynamiques (comme celles formées par `.query()`, `.insert()`, `.update()`, ou `.delete()`), ce qui limite son utilité pour ces types de requêtes qui dominent notre code.

Sous-schéma logique

Pour répondre à la question visant à dériver le Logical SubSchema (LSS), il est nécessaire d'identifier les éléments du schéma (tables et colonnes) réellement exploités par les programmes. Le LSS est obtenu en excluant toutes les parties inutilisées du schéma logique initial (LS), de manière à refléter uniquement les interactions effectives entre le code source et la base de données.

La méthodologie suivie pour cette analyse est la suivante :

1. Identifier toutes les tables définies dans le schéma logique initial (LS).
2. Pour chaque table, examiner le code source afin de localiser toutes les interactions, en particulier :
3. Les opérations utilisant des curseurs (`Cursor`) pour extraire des colonnes spécifiques via des méthodes telles que `getColumnIndexOrThrow`.
4. Analyser les méthodes associées aux curseurs et aux requêtes pour déterminer où elles sont appelées dans le programme, et identifier les colonnes réellement utilisées pour chaque table.

5. Construire le Logical SubSchema (LSS) en conservant uniquement les tables et colonnes utilisées par le programme.

Cette approche garantit une vue précise des éléments du schéma qui sont essentiels au fonctionnement du programme, tout en éliminant les parties redondantes ou inutilisées. Il est important de noter que, bien que les méthodes comme `.query()` insèrent majoritairement toutes les colonnes d'une table dans le curseur (`SELECT *`) dans notre programme, nous avons pris le parti de ne pas simplement considérer ces colonnes comme utilisées. Une vérification plus approfondie a été effectuée pour déterminer si les colonnes récupérées dans les curseurs sont réellement exploitées dans le programme. Cela garantit que le LSS reflète uniquement les interactions effectives et significatives entre le code source et la base de données.

Table Account

La classe `Account` utilise toutes les colonnes de la table `Account` dans la méthode `Account.fromCursor(Cursor cursor)`. Les colonnes suivantes sont utilisées :

- `UUID`
- `USERNAME`
- `SERVER`
- `PASSWORD`
- `OPTIONS`
- `ROSTERVERSION`
- `KEYS`
- `AVATAR`
- `DISPLAY_NAME`
- `HOSTNAME`
- `PORT`
- `STATUS`
- `STATUS_MESSAGE`
- `RESOURCE`
- `PINNED_MECHANISM`
- `PINNED_CHANNEL_BINDING`
- `FAST_MECHANISM`
- `FAST_TOKEN`

Ainsi, la méthode `fromCursor` garantit une couverture complète des colonnes de la table `Account`.

Table Contact

La classe `Contact` utilise toutes les colonnes de la table `Contact` dans la méthode `Contact.fromCursor(Cursor cursor)`. Les colonnes suivantes sont utilisées :

- `ACCOUNT`
- `SYSTEMNAME`

— SERVERNAME
— PRESENCE_NAME
— JID
— OPTIONS
— SYSTEMACCOUNT
— PHOTOURI
— KEYS
— AVATAR
— LAST_PRESENCE
— LAST_TIME
— GROUPS
— RTP_CAPABILITY

Ainsi, la méthode `fromCursor` garantit une couverture complète des colonnes de la table `Contact`.

Table Conversation

La classe `Conversation` utilise toutes les colonnes de la table `Conversation` dans la méthode `Conversation.fromCursor(Cursor cursor)`. Les colonnes suivantes sont utilisées :

— UUID
— NAME
— CONTACT
— ACCOUNT
— CONTACTJID
— CREATED
— STATUS
— MODE
— ATTRIBUTES

Ainsi, la méthode `fromCursor` garantit une couverture complète des colonnes de la table `Conversation`.

Table PresenceTemplate

La classe `PresenceTemplate` utilise toutes les colonnes de la table `PresenceTemplate` dans la méthode `PresenceTemplate.fromCursor(Cursor cursor)`. Les colonnes suivantes sont utilisées :

— UUID
— MESSAGE
— STATUS

Ainsi, la méthode `fromCursor` garantit une couverture complète des colonnes de la

table `PresenceTemplate` sauf pour `last_used` car la variable capturant cette colonne est private et n'est donc pas utilisable dans le code et il n'y a pas de getter ni setter pour cette variable.

Table Message

La classe `Message` utilise toutes les colonnes de la table `Message` dans la méthode `Message.fromCursor(Cursor cursor, Conversation conversation)`. Les colonnes suivantes sont utilisées :

- `UUID`
- `CONVERSATION`
- `COUNTERPART`
- `TRUE_COUNTERPART`
- `BODY`
- `TIME_SENT`
- `ENCRYPTION`
- `STATUS`
- `TYPE`
- `CARBON`
- `REMOTE_MSG_ID`
- `RELATIVE_FILE_PATH`
- `SERVER_MSG_ID`
- `FINGERPRINT`
- `READ`
- `EDITED`
- `OOB`
- `ERROR_MESSAGE`
- `READ_BY_MARKERS`
- `MARKABLE`
- `DELETED`
- `BODY_LANGUAGE`
- `OCCUPANT_ID`
- `REACTIONS`

Ainsi, la méthode `fromCursor` garantit une couverture complète des colonnes de la table `Message`.

Tables prekeys et signed_prekeys

Les classes utilisant les tables `prekeys` et `signed_prekeys` garantissent une couverture des colonnes suivantes :

Table prekeys Les colonnes de la table `prekeys` sont utilisées dans les méthodes associées au `PreKeyStore` :

— `ACCOUNT` : Identifiant de l'utilisateur associé à la clé.
— `ID` : Identifiant unique de la clé.
— `KEY` : Valeur de la clé prépartagée.

Méthodes impliquées dans l'accès à la table `prekeys` :

— `SQLiteAxolotlStore.loadPreKey(int preKeyId)` : Charge un enregistrement de clé prépartagée.
— `SQLiteAxolotlStore.storePreKey(int preKeyId, PreKeyRecord record)` : Stocke un enregistrement de clé prépartagée.
— `SQLiteAxolotlStore.containsPreKey(int preKeyId)` : Vérifie l'existence d'une clé prépartagée.
— `SQLiteAxolotlStore.removePreKey(int preKeyId)` : Supprime une clé prépartagée.
— `SQLiteAxolotlStore.flushPreKeys()` : Supprime les clés marquées pour suppression.

Table signed_prekeys Les colonnes de la table `signed_prekeys` sont utilisées dans les méthodes associées au `SignedPreKeyStore` :

— `ACCOUNT` : Identifiant de l'utilisateur associé à la clé.
— `ID` : Identifiant unique de la clé signée.
— `KEY` : Valeur de la clé signée.

Méthodes impliquées dans l'accès à la table `signed_prekeys` :

— `SQLiteAxolotlStore.loadSignedPreKey(int signedPreKeyId)` : Charge un enregistrement de clé signée.
— `SQLiteAxolotlStore.storeSignedPreKey(int signedPreKeyId, SignedPreKeyRecord record)` : Stocke un enregistrement de clé signée.
— `SQLiteAxolotlStore.containsSignedPreKey(int signedPreKeyId)` : Vérifie l'existence d'une clé signée.
— `SQLiteAxolotlStore.removeSignedPreKey(int signedPreKeyId)` : Supprime une clé signée.
— `SQLiteAxolotlStore.loadSignedPreKeys()` : Charge toutes les clés signées.

Table identities

La table `identities` est utilisée pour stocker et gérer les identités Signal Protocol. Les colonnes définies dans cette table sont les suivantes :

— `ACCOUNT` : Identifiant de l'utilisateur associé.
— `NAME` : Nom associé à l'identité.
— `OWN` : Indicateur si l'identité appartient à l'utilisateur local.
— `FINGERPRINT` : Empreinte unique de la clé d'identité.
— `CERTIFICATE` : Certificat X509 associé.
— `TRUST` : Niveau de confiance pour cette identité.
— `ACTIVE` : Indicateur si l'identité est active.
— `LAST_ACTIVATION` : Dernière date d'activation de l'identité.
— `KEY` : Valeur de la clé publique associée.

Utilisation des colonnes via un curseur :

- **Méthode** `FingerprintStatus.fromCursor(Cursor cursor)` :
  - Colonnes utilisées : `TRUST`, `ACTIVE`, `LAST_ACTIVATION`.
- **Méthode** `getFingerprintStatus` :
  - Colonnes utilisées : `FINGERPRINT`, `TRUST`.
- **Méthode** `setFingerprintStatus` :
  - Colonnes utilisées : `FINGERPRINT`, `TRUST`, `ACTIVE`.
- **Méthode** `setFingerprintCertificate` :
  - Colonnes utilisées : `FINGERPRINT`, `CERTIFICATE`.
- **Méthode** `loadIdentityKeys` :
  - Colonnes utilisées : `ACCOUNT`, `NAME`, `FINGERPRINT`, `TRUST`, `OWN`, `KEY`.
- **Méthode** `storeIdentityKey` :
  - Colonnes utilisées : `ACCOUNT`, `NAME`, `FINGERPRINT`, `TRUST`.

Colonnes non utilisées :

Toutes les colonnes sont utilisées.

Table sessions

La table `sessions` est utilisée pour gérer les sessions Signal Protocol. Les colonnes définies dans cette table sont les suivantes :

- `ACCOUNT` : Identifiant de l'utilisateur associé à la session.
- `NAME` : Nom du correspondant.
- `DEVICE_ID` : Identifiant unique du périphérique.
- `KEY` : Données de la session.

Utilisation des colonnes via un curseur :

- **Méthode** `loadSession` :
  - Colonnes utilisées : `ACCOUNT`, `NAME`, `DEVICE_ID`, `KEY`.
- **Méthode** `storeSession` :
  - Colonnes utilisées : `ACCOUNT`, `NAME`, `DEVICE_ID`, `KEY`.
- **Méthode** `containsSession` :
  - Colonnes utilisées : `ACCOUNT`, `NAME`, `DEVICE_ID`.
- **Méthode** `deleteSession` :
  - Colonnes utilisées : `ACCOUNT`, `NAME`, `DEVICE_ID`.

Colonnes non utilisées :

- Toutes les colonnes sont utilisées.

Table PUSH

La table `PUSH` est utilisée pour gérer les notifications push dans l'application. Les colonnes définies dans cette table sont les suivantes :

- `account` : Identifiant de l'utilisateur associé.
- `transport` : Type de transport utilisé pour la notification push.
- `application` : Nom de l'application.
- `instance` : Instance unique (valeur unique dans la table).
- `endpoint` : URL de l'endpoint pour la notification push.
- `expiration` : Timestamp de l'expiration de l'entrée.

Utilisation des colonnes via un curseur :

- **Méthode** `register` :
  - Colonnes utilisées : `application`, `instance`.
- **Méthode** `getRenewals` :
  - Colonnes utilisées : `application`, `instance`, `account`, `transport`, `expiration`.
- **Méthode** `getEndpoint` :
  - Colonnes utilisées : `account`, `transport`, `instance`, `endpoint`, `expiration`.
- **Méthode** `deletePushTargets` :
  - Colonnes utilisées : `application`, `instance`.
- **Méthode** `hasEndpoints` :
  - Colonnes utilisées : `endpoint`.
- **Méthode** `updateEndpoint` :
  - Colonnes utilisées : `instance`, `account`, `transport`, `endpoint`, `expiration`.
- **Méthode** `getPushTargets` :
  - Colonnes utilisées : `application`, `instance`, `account`.
- **Méthode** `deleteInstance` :
  - Colonnes utilisées : `instance`.
- **Méthode** `deleteApplication` :
  - Colonnes utilisées : `application`.

Colonnes non utilisées :

- Toutes les colonnes définies sont utilisées.

Table Discovery Results

La table `discovery_results` est utilisée pour stocker les résultats de découverte de services. Les colonnes définies dans cette table sont les suivantes :

- `HASH` : Hachage unique associé au résultat de découverte.
- `VER` : Version ou identifiant du résultat.
- `RESULT` : Résultat de la découverte, stocké en tant que chaîne JSON.

Utilisation des colonnes via un curseur :
- **Méthode** `ServiceDiscoveryResult(Cursor cursor)` :
  - Colonnes utilisées : `HASH`, `VER`, `RESULT`.

Colonnes non utilisées :
- Toutes les colonnes définies sont utilisées.

Table Resolver Results

La table `resolver_results` est utilisée pour stocker les résultats de résolution DNS. Les colonnes définies dans cette table sont les suivantes :

- `DOMAIN` : Domaine pour lequel la résolution est effectuée.
- `HOSTNAME` : Nom d'hôte obtenu pour le domaine.
- `IP` : Adresse IP correspondante.
- `PRIORITY` : Priorité du résultat.
- `DIRECT_TLS` : Indique si TLS direct est activé.
- `AUTHENTICATED` : Indique si la connexion est authentifiée.
- `PORT` : Port associé à la connexion.

Utilisation des colonnes via un curseur :
- **Méthode** `findResolverResult` :
  - Colonnes utilisées : `DOMAIN`, `HOSTNAME`, `IP`, `PRIORITY`, `DIRECT_TLS`, `AUTHENTICATED`, `PORT`.
  - Cette méthode est utilisée dans le fichier `XmppConnection.java`.
- **Méthode** `saveResolverResult` :
  - Colonnes utilisées : `DOMAIN`, `HOSTNAME`, `IP`, `PRIORITY`, `DIRECT_TLS`, `AUTHENTICATED`, `PORT`.
  - Cette méthode est utilisée dans le fichier `XmppConnection.java`.

Colonnes non utilisées :
- Toutes les colonnes définies sont utilisées.

Conclusion : Après une analyse approfondie du code on a montré que toutes les colonnes et tables sont utilisées dans le code sauf `PresenceTemplate.last_used`.

Étape 3 : Analyse « What-If » et Scénarios d'Évolution

Introduction

Dans cette étape, nous appliquons une analyse « what-if » pour évaluer l'impact de 10 scénarios d'évolution sur le Sous-Schéma Physique (PSS) de notre système de base de données. Chaque scénario est accompagné d'une description détaillée et de son impact sur le code source, incluant les classes, méthodes, et requêtes SQL affectées.

Scénario 1 : Ajout de contrainte dans le moteur de base de données

Résumé du changement :

Lors de notre analyse du code Java et DDL de l'application, nous avons remarqué que certaines colonnes dans les tables de la base de données peuvent contenir des valeurs nulles et d'autres ne sont plus utilisées . Cependant, le code Java ne semble pas gérer ces valeurs nulles. Pour éviter des erreurs potentielles, nous allons ajouter des contraintes *NOT NULL* sur ces colonnes.

Détail des changements :

Colonnes à modifier :

- Dans la table `contacts`, la colonne `jid` peut être `NULL`. Nous allons ajouter une contrainte `NOT NULL` pour garantir que chaque contact a un identifiant Jabber valide.
- Dans la table `accounts`, les colonnes `username` et `server` peuvent être `NULL`. Nous allons ajouter une contrainte `NOT NULL` pour garantir que chaque compte possède un server et un username. Ce qui revient à dire que chaque compte doit avoir un identifiant Jabber valide.
- Dans la table `conversation`, la colonne `contactUuid` ne semble plus réutilisée car ne référence pas une colonne `uuid` dans la table `contact`. Nous allons supprimer cette colonne. La colonne `contactJid` ne dois aussi pas être null. De plus, cette colonne et la colonne `contactUuid` doivent être unique.

Il est quand même sage de vérifier que les colonnes à modifier ne contiennent pas de valeurs nulles avant d'appliquer les contraintes `NOT NULL`.

Par simplicité, nous allons supprimer les instances qui possèdent des valeurs nulles pour les colonnes où on ajoute la contrainte `NOT NULL`. Mais il faudrait faire un backup de la base de données avant d'appliquer ces changements.

```
DELETE FROM contacts WHERE jid IS NULL;
ALTER TABLE contacts ALTER COLUMN jid SET NOT NULL;

DELETE FROM accounts WHERE username IS NULL OR server IS NULL;
ALTER TABLE accounts ALTER COLUMN username SET NOT NULL;
```

```
ALTER TABLE accounts ALTER COLUMN server SET NOT NULL;

DELETE FROM conversation WHERE jid IS NULL;
ALTER TABLE conversation DROP COLUMN contactUuid;
ALTER TABLE conversation ALTER COLUMN contactJid SET NOT NULL;
ALTER TABLE conversation ADD UNIQUE(contactJid, accountUuid)
 ON CONFLICT IGNORE;
```

Il est maintenant possible de déclarer explicitement la clé étrangère `contactJid` dans la table `Conversation`.

```
ALTER TABLE conversation ADD FOREIGN KEY(contactJid)
 REFERENCES contact(jid) ON DELETE CASCADE;
```

On notera que dans la classe `Contact`, seule la colonne `jid` profite de l'ajout de la contrainte `NOT NULL`. La colonne `accountUuid` n'en profite pas, car cela ne poserait pas de problème avec le code actuel de l'application. Malgré tout, il est nécessaire d'avoir à la fois un `jid` et un `accountUuid` pour identifier un contact.

Impact sur le code :

L'idée de ce scénario est de s'assurer que les colonnes qui ne doivent pas être `NULL` ne le sont pas. Le code Java ne va pas être directement impacté par ces changements. Mais il faut tout de même faire tourner les tests pour s'assurer que l'application fonctionne toujours correctement.

Test à effectuer :

- Vérifier que les colonnes modifiées ne contiennent pas de valeurs nulles.
- Appliquer les contraintes `NOT NULL` et vérifier que les valeurs nulles sont supprimées.
- Tester les fonctionnalités principales de l'application pour s'assurer qu'elles fonctionnent toujours correctement.

Scénario 2 : Ajout d'une fonctionnalité d'appel

Résumé du changement :

Le protocole XMPP vient de se mettre à jour et permet maintenant de gérer des appels. Nous allons ajouter une fonctionnalité d'appel à l'application, en utilisant les fonctionnalités de base de l'application pour gérer les appels.

Détail des changements :

Stratégie d'implémentation :

La majorité des modifications nécessaires est liée au code Java. La base de données doit s'adapter à ces changements. Ici, nous sommes dans un scénario imaginaire, mais on peut imaginer qu'il faut ajouter une table `calls` pour stocker les informations sur les appels. Cette table contiendrait des informations sur le temps de début et de fin de l'appel, la durée, etc. Elle sera en relation avec la table conversation, ce qui permet de minimiser la quantité de données à stocker et de faciliter les requêtes.

Mais aussi de nouvelles colonnes techniques permettant d'effectuer des appels dans les tables `Account` et `Contact`. Par exemple, une colonne `voiceSecurityKey` dans la table `Account` pour stocker le mécanisme de chiffrement utilisé pour les appels.

Modification :

Dans un premier temps , nous allons ajouter une table `calls` pour stocker les informations sur les appels.

```
CREATE TABLE Calls (
        uuid TEXT PRIMARY KEY,
        conversationUuid TEXT,
        startTime NUMBER,
        duration NUMBER,
        status NUMBER,

        FOREIGN KEY(conversationUuid)
         REFERENCES conversation(uuid) ON DELETE CASCADE
);
```

Ensuite, nous allons ajouter des colonnes dans les tables `Account` et `Contact` pour stocker les informations nécessaires pour les appels.

```
ALTER TABLE accounts ADD COLUMN voiceSecurityKey TEXT;
ALTER TABLE contacts ADD COLUMN voiceSecurityKey TEXT;
```

Impact sur le code :

L'ajout d'une telle fonctionnalité nécessite des modifications importantes dans le code Java. Il faudra ajouter des classes, des méthodes, des requêtes SQL, etc. pour gérer les appels. Il faudra aussi mettre à jour l'interface utilisateur pour permettre aux utilisateurs de passer des appels.

Voici une liste non exhaustive des modifications à apporter dans le code Java :

- Ajout d'un package, ou tout du moins d'une classe, contenant la logique pour gérer les appels.
  - Possibilité d'ajouter des appels à la base de données à la fin de l'appel.
  - Permettre de démarrer et de terminer un appel.
  - Lister l'historique des appels. Exemple en utilisant des jointures mais bien sur, on préférera l'utilisation de l'ORM :
    ```
    SELECT Calls.*, Contacts.presence_name FROM Calls
    JOIN Conversation ON Calls.conversationUuid = Conversation.uuid
    JOIN Accounts ON Conversation.accountUuid = Accounts.uuid
    JOIN Contacts ON Conversation.contactJid = Contacts.jid
    AND Conversation.accountUuid = Contacts.accountUuid
    WHERE Accounts.uuid = 'target_account_uuid';
    ```
- Ajout d'un nouveau statut possible pour définir un template de présence. Celui-ci pourrait être `in-call`.
- Mise à jour de l'interface utilisateur pour permettre aux utilisateurs de passer des appels.

Test à effectuer :

- Faire des tests unitaires pour s'assurer que les appels sont correctement ajoutés à la base de données.
  - Que ce passerai t-il si l'appel n'a pas de conversation associée (valeur Null).
  - Que ce passerai t-il si aucune clé de sécurité a été définie pour l'appel.
  - Ajout et suppression d'appel.
- Tester les fonctionnalités d'appel pour s'assurer qu'elles fonctionnent correctement.

Scénario 3 : Ajout de la possibilité de lier un numéro de téléphone à un contact

Résumé du changement :

XMPP s'unifie avec les protocoles de messagerie traditionnels et permet maintenant d'envoyer des messages SMS et d'appeler des numéros de téléphone. Le numéro de téléphone remplace le *jid*, il n'est donc plus question de *jid* dans XMPP.

Nous allons ajouter la possibilité de lier un numéro de téléphone à un contact dans l'application. Et remplacer le *jid* par le numéro de téléphone dans les tables `contacts` et `Account`.

Dans ce scénario, nous allons ignorer que certains contacts peuvent être des groupes, et donc ne pas avoir de numéro de téléphone.

Détail des changements :

Stratégie d'implémentation :

Dans le cas de la table `Account`, il est possible de remplacer le *jid* par le numéro de téléphone. Cependant, dans les tables `Contacts`, le jid est utilisé comme pour une relation avec la table `Conversation` et `Message`. Il faudra donc modifier adéquatement les tables `Conversation` et `Message` pour utiliser le numéro de téléphone à la place du *jid*.

Pour le code Java, une solution permettant de ne pas modifier le code en profondeur est possible dans le cas où le `jid` est simplement remplacé par le numéro de téléphone. On peut ajouter une méthode dans l'ORM permettant de récupérer les numéros de téléphone dans les différentes classes, tout en gardant les méthodes `getJid` qui serai un alias de `getPhoneNumber`. On devra, malgré tout, marquer le `getJid` comme `deprecated` afin éviter des erreurs dans le futur.

Modification :

Dans un premiers temps nous allons ajouter le numéro de téléphone dans les tables `Account` et `Contacts`.

```
ALTER TABLE accounts ADD COLUMN phoneNumber TEXT;
ALTER TABLE contacts ADD COLUMN phoneNumber TEXT;
```

Ensuite, il faudra attendre que l'utilisateur ajoute les numéro de téléphone, aussi bien le sien que celui de ses contacts.

Dès que cela est fait, il faudra aussi modifier les tables `Conversation` et `Message` pour utiliser le numéro de téléphone à la place du *jid*.

Pour la table `Conversation` :

```
ALTER TABLE conversation
ADD COLUMN contactPhoneNumber TEXT;

UPDATE conversation
SET contactPhoneNumber = (
        SELECT phoneNumber FROM contacts
        WHERE jid = contactJid
);

ALTER TABLE conversation DROP COLUMN contactJid;

ALTER TABLE conversation
ADD UNIQUE(contactPhoneNumber, accountUuid)
ON CONFLICT IGNORE;
```

Et pour la table `Message` :

```
UPDATE message
WHERE trueCounterpart IS NOT NULL
SET trueCounterpart = (
        SELECT phoneNumber FROM contacts
        WHERE jid = trueCounterpart
);

UPDATE message
WHERE counterpart IS NOT NULL
SET counterpart = (
        SELECT phoneNumber FROM contacts
        WHERE jid = counterpart
);
```

Impact sur le code :

L'impact sur le code Java est relativement faible, car il suffit de remplacer le *jid* par le numéro de téléphone dans les classes qui manipulent les contacts. Il faudra aussi mettre à jour l'interface utilisateur pour permettre aux utilisateurs de saisir et de gérer les numéros de téléphone.

Il faudra aussi concevoir un script permettant à l'utilisateur de migrer ses contacts de *jid* à numéro de téléphone, et puis lancer les requêtes SQL pour effectuer la migration.

Scénario 4 : Préférence de compte

Résumé du changement :

Nous allons ajouter une fonctionnalité de préférence de compte à l'application. Cela permettra aux utilisateurs de définir des préférences pour chaque compte, telles que le thème, la langue, etc.

Détail des changements :

Stratégie d'implémentation :

Pour ajouter cette fonctionnalité, nous allons ajouter une nouvelle table `accountPreferences` pour stocker les préférences de compte. Cette table sera en relation avec la table `Account`. Cela permettra à l'utilisateur de sauvegarder des configurations de préférences pour un même compte.

Modification :

Nous allons ajouter une table `preferences` pour stocker les préférences de compte. Cette table sera en relation avec la table `Account`.

```
CREATE TABLE accountPreferences (
        uuid TEXT PRIMARY KEY,
        accountUuid TEXT,
        theme TEXT,
        language TEXT,
        FOREIGN KEY(accountUuid)
        REFERENCES accounts(uuid) ON DELETE CASCADE
);
```

Impact sur le code :

La plupart des modifications nécessaires pour ajouter cette fonctionnalité seront dans le code Java. Il faudra ajouter des classes, des méthodes, des requêtes SQL, etc. pour gérer les préférences de compte. Il faudra aussi mettre à jour l'interface utilisateur pour permettre aux utilisateurs de définir et de gérer leurs préférences.

Il faudra certainement utiliser des outils d'internationalisation pour gérer les préférences de langue.

Scénario 5 : Possibilité de contacter une intelligence artificielle

Résumé du changement :

Nous allons ajouter une fonctionnalité permettant aux utilisateurs de contacter une intelligence artificielle. Cela permettra aux utilisateurs de poser des questions à l'intelligence artificielle et de recevoir des réponses.

Détail des changements :

Stratégie d'implémentation :

Pour ajouter cette fonctionnalité, nous allons ajouter une nouvelle table `chatbot` pour stocker les informations sur les différentes intelligences artificielles. Cette table nous oblige à modifier la table `Contact` pour permettre de lier un contact à une intelligence artificielle.

Modification :

Nous allons ajouter une table `chatbot` pour stocker les informations sur les intelligences artificielles. Cette table sera en relation avec la table `Contact`.

Ce choix permet de simuler une conversation avec une intelligence artificielle en utilisant les mêmes fonctionnalités que pour une conversation normale. Cependant, il faudra modifier la table `Contact` pour diviser cette classe en deux classes, une pour les contacts en général et une pour les personnes réelles.

Création de la table `chatbot` :

```
CREATE TABLE chatbot (
        uuid TEXT PRIMARY KEY,
        contactUuid TEXT,
        name TEXT,
        serviceURL TEXT,
        endpoint TEXT,
        description TEXT,
        FOREIGN KEY(contactUuid)
        REFERENCES contacts(uuid) ON DELETE CASCADE,
        UNIQUE (contactUuid) ON CONFLICT IGNORE
);
```

Création de la table `person` :

```
CREATE TABLE person (
        uuid TEXT PRIMARY KEY,
        contactUuid TEXT,
        servername TEXT,
        jid TEXT,
        systemaccount NUMERIC
        last_presence TEXT,
        last_time NUMERIC,

        FOREIGN KEY(contactUuid)
        REFERENCES contacts(uuid) ON DELETE CASCADE,
        UNIQUE (jid, contactUuid) ON CONFLICT IGNORE
);
```

Modification de la table `contact` :

```
ALTER TABLE contact RENAME TO contactTemp;
CREATE TABLE contact (
        accountUuid TEXT
        systemname TEXT,
        presence_name TEXT,
        pgpkey TEXT,
        photouri TEXT,
        options NUMERIC,
```

```
        avatar TEXT,
        rtpCapability TEXT,
        groups TEXT,

        FOREIGN KEY(accountUuid)
        REFERENCES accounts(uuid) ON DELETE CASCADE
);
```

Il faut aussi ajouter des triggers afin d'éviter qu'une personne et qu'un chatbot soient liés à un même contact.

```
CREATE TRIGGER chatbot_trigger
BEFORE INSERT ON chatbot
BEGIN
        IF EXISTS (
                SELECT 1 FROM person
                WHERE contactUuid = NEW.contactUuid
        ) THEN
                RAISE(
                ABORT,
                'Contact already exists in person table'
                );
        END IF;
END;

CREATE TRIGGER person_trigger
BEFORE INSERT ON person
BEGIN
        IF EXISTS (
                SELECT 1 FROM chatbot
                WHERE contactUuid = NEW.contactUuid
        ) THEN
                RAISE(
                ABORT,
                'Contact already exists in chatbot table'
                );
        END IF;
END;
```

Il faut aussi garantir la contrainte de `UNIQUE` sur la colonne `jid` de la table `person` et la colonne `accountUuid` de la table `contact`. Pour cela, on utilise un trigger pour vérifier que la contrainte est respectée.

```
CREATE TRIGGER unique_jid_accountUuid_constraint
BEFORE INSERT ON person
FOR EACH ROW
BEGIN
        SELECT RAISE(
        ABORT,
        'Duplicate jid for the same accountUuid'
```

```
        )
        WHERE EXISTS (
        SELECT 1
        FROM person
        JOIN contacts ON person.contactUuid = contacts.uuid
        WHERE person.jid = NEW.jid
        AND contacts.accountUuid = (
                SELECT accountUuid
                FROM contacts
                WHERE uuid = NEW.contactUuid
        ));
END;
```

Impact sur le code :

Il faudra dans le code Java ajouter des classes, des méthodes, des requêtes SQL, etc. pour gérer les conversations avec les intelligences artificielles. Il faudra aussi mettre à jour l'interface utilisateur pour permettre aux utilisateurs de contacter les intelligences artificielles.

Pour cela, il faudra en premier tant modifier les classes en relation avec l'ORM. C'est-à-dire, *eu.siacs.conversations.persistance.DatabaseBackend* doit intégrer dans sa méthode `onCreate` la création des tables `chatbot` et `person`. Il faudra aussi modifier la méthode `onUpgrade` pour gérer les changements de version de la base de données.

Ensuite, la classe correspondante à la table `Contact` doit être modifiée pour représenter cette nouvelle structure. Donc, dans *eu.siacs.conversations.entities.Contact*, il faudra retirer les attributs correspondants aux colonnes retirées. Par exemple, servername, jid, etc. Il faudra aussi ajouter des méthodes permettant de gérer les nouveaux attributs. Par exemple, `getChatbot` et `getPerson`.

Pour finir, il faudra ajouter des classes permettant de gérer les conversations avec les intelligences artificielles. *eu.siacs.conversations.entities.Chatbot* et *eu.siacs.conversations.entities.Person* reprendront les attributs de la table `Chatbot` et `Person` respectivement, et seront implémentées de la même manière que les autres classes du package `entities`.

Il est bien sûr à noter que les activités responsables de la gestion des contacts devront être modifiées pour gérer les nouveaux types de contacts. Par exemple *eu.siacs.conversations.ui.ContactDetailsActivity* doit prendre en compte qu'un contact peut être une personne avec un `JID`, mais peut être un bot ne possédant pas de `JID`.

Scénario 6 : Suppression des données de sécurité

Résumé du changement :

Avec l'arrivée des ordinateurs quantiques, les données de sécurité actuelles ne sont plus considérées comme nécessaires. Nous allons supprimer les données et structures associées à la sécurité dans le système.

Détail des changements :

Tables à supprimer :

- `identities`, `signed_prekeys`, et `prekeys` : ces tables stockent des données liées aux clés cryptographiques.

Exemple de requête SQL :

```
DROP TABLE identities;
DROP TABLE signed_prekeys;
DROP TABLE prekeys;
```

Colonnes à supprimer :

- Dans la table `accounts`, les colonnes suivantes seront supprimées :
  - `pinned_mechanism`
  - `fast_mechanism`

Exemple de requête SQL :

```
ALTER TABLE accounts DROP COLUMN pinned_mechanism;
ALTER TABLE accounts DROP COLUMN fast_mechanism;
```

Impact sur le code :

- Les méthodes manipulant ces données doivent être retirées. Exemple :
  - Méthodes dans `UnifiedPushDatabase` et `DatabaseBackend`.
  - Ajouter leur suppression dans la méthode `onUpgrade` :

```
if (oldVersion < NEW_VERSION && newVersion >= NEW_VERSION) {
    db.execSQL("DROP TABLE IF EXISTS identities");
    db.execSQL("DROP TABLE IF EXISTS signed_prekeys");
    db.execSQL("DROP TABLE IF EXISTS prekeys");
}
```

- Les API ou modules front-end utilisant ces données doivent être refactorisés pour fonctionner sans elles.

Test à effectuer :

— Vérifier que les données de sécurité sont bien supprimées sans erreur.
— Confirmer que les fonctionnalités principales restent opérationnelles.

—

Scénario 7 : Renommer certaines colonnes et tables

Résumé du changement :

Pour rendre le schéma plus clair, nous renommons certaines tables et colonnes pour qu'elles soient compréhensibles par de nouveaux collaborateurs.

Détail des changements :

Renommer la table `conversations` en `chats` : Cela implique de modifier toutes les références dans les requêtes SQL, ainsi que dans le code source.

Exemple de requête SQL :

```
ALTER TABLE conversations RENAME TO chats;
```

Renommer la colonne `body` dans `messages` en `content` : Cette colonne est utilisée pour stocker le contenu des messages. Le nouveau nom est plus explicite.

Exemple de requête SQL :

```
ALTER TABLE messages RENAME COLUMN body TO content;
```

Impact sur le code :

— Mettre à jour les méthodes d'accès à la base de données, telles que :
  — Les jointures impliquant `conversations`/`chats`.
  — Les sélections ou inserts utilisant `messages.body`.
— Ajouter les changements dans la méthode `onUpgrade`

Exemple de changement dans le code Java :

```
Avant :
db.query("conversations", null, "name=?", new String[] {chatName}, null, null,

Apres :
db.query("chats", null, "name=?", new String[] {chatName}, null, null, null);
```

Test à effectuer :

— Tester toutes les fonctionnalités qui utilisent `conversations` ou `messages.body`.
— Valider que les données migrées sont intactes.

—

Scénario 8 : Ajouter des catégories de messages

Résumé du changement :

Ajouter une colonne `category` dans la table `messages` pour permettre de classer les messages en catégories (ex. multimédia, GPS, etc.).

Détail des changements :

Ajouter une nouvelle colonne : Cette colonne permettra d'identifier le type de chaque message.

Exemple de requête SQL :

```
ALTER TABLE messages ADD COLUMN category TEXT;
```

Adapter les requêtes SQL : Lors de l'insertion d'un message, la catégorie doit être précisée :

```
ContentValues values = new ContentValues();
values.put("category", "multimedia");
db.insert("messages", null, values);
```

Impact sur le code

— Modifier la classe `Message` pour inclure l'attribut `category`.
— Ajouter un support dans les méthodes d'insertion et de récupération des messages.

Interface utilisateur et API :

— Ajouter une logique dans l'interface utilisateur pour afficher les catégories et permettre le filtrage.

Test à effectuer :

— Ajouter différents types de messages et vérifier que la catégorie est correctement stockée.
— Valider que les messages sont affichés correctement par catégorie.

—

Scénario 9 : Envoi automatique de message lors de l'ajout d'un contact

Résumé du changement :

Lors de l'ajout d'un nouveau contact, un message de bienvenue est automatiquement envoyé.

Détail des changements :

Ajouter un trigger SQL : Le trigger détecte l'insertion dans la table `contacts` et insère un message dans `messages`.

Exemple de requête SQL :

```
CREATE TRIGGER add_contact_message
AFTER INSERT ON contacts
BEGIN
    INSERT INTO messages (conversationUuid, body, type, timeSent)
    VALUES (NEW.uuid, 'Bienvenue !', 1, strftime('%s', 'now'));
END;
```

Impact sur le code :

— Les méthodes de gestion des contacts n'ont pas besoin de modification directe, mais il faut valider que le trigger fonctionne correctement.

Test à effectuer :

— Ajouter un contact et vérifier que le message de bienvenue est automatiquement créé.
— Vérifier que les anciens contacts ne déclenchent pas le trigger.

—

Scénario 10 : Répondre à un message

Résumé du changement :

Ajouter une fonctionnalité permettant de répondre à un message existant, en liant les messages sous forme de threads.

Détail des changements :

Ajouter une colonne `reply_to` : Cette colonne stocke l'identifiant du message auquel on répond.

Exemple de requête SQL :

```
ALTER TABLE messages ADD COLUMN reply_to TEXT;
```

Adapter les requêtes SQL : Lors de l'insertion, indiquer si un message est une réponse :

```
ContentValues values = new ContentValues();
values.put("reply_to", repliedMessageUuid);
db.insert("messages", null, values);
```

Pour récupérer les messages dans un thread :

```
SELECT * FROM messages WHERE reply_to = 'message_uuid';
```

Interface utilisateur et API :

- Modifier l'interface utilisateur pour afficher les messages sous forme de threads.
- Ajouter une option "Répondre" dans le menu contextuel des messages.

Test à effectuer :

- Répondre à différents messages et vérifier que les liens (`reply_to`) sont correctement créés.
- Valider l'affichage des threads dans l'interface utilisateur.

Étape 4 : Qualité de la base de données

Introduction

Après avoir analysé la base de donnée, nous pouvons donc maintenant établir une liste des bonnes réalisations et des mauvaises pratiques. Pour chaque mauvaise pratique, nous allons proposer une solution pour corriger le problème.

L'application des correctifs permettra à la base de données d'évoluer de manière plus saine et plus robuste. Cela permettra aussi de faciliter l'ajout de nouvelles fonctionnalités et la contribution extérieure, tout en réduisant les coûts de maintenance.

Bonnes réalisations

La base de données présente plusieurs bonnes réalisations, notamment :

— **Centralisation de la gestion de la base de données** : Dans le code Java, la gestion de la base de données est centralisée principalement dans les classes `DatabaseBackend` et `UnifiedPushDatabase`. Cela permet de faciliter la modification de comportement de la base de données. Par exemple, si on veut changer le système de gestion de la base de données, il suffit de modifier ces classes.
— **Utilisation d'une nomenclature cohérente** : Les noms des tables et des colonnes sont explicites et suivent une nomenclature cohérente. Par exemple, les tables sont nommées en anglais et font directement référence à leur semantique. Elle suit aussi le lexique utilisé à travers le protocole XMPP. Ceci permet de faciliter la compréhension de la base de données.
— **Versionnage** : La base de données présente un gestionnaire de version. Ce qui permet à la base de données de se mettre à jour d'une version antérieure à une version plus récente. Cela permet de garantir la cohérence des données et de faciliter l'évolution de la base de données.
— **Utilisation d'un UUID** : Dans la plupart des tables, un UUID est utilisé comme clé primaire. Cela permet d'éviter les problèmes de collision de clés et de garantir l'unicité des enregistrements. Cela permet aussi de faciliter la gestion des relations entre les tables.

Mal façons

La base de données présente également quelques mauvaises pratiques, notamment :

Clé étrangère implicite

Comme on l'a déjà mentionné dans l'analyse des schémas, la base de données contient des clés étrangères implicites. Cela peut poser des problèmes d'intégrité référentielle et de

cohérence des données. Il est donc recommandé d'ajouter des clés étrangères explicites pour garantir l'intégrité référentielle des données.

On pense à la relation entre les tables `Conversation` et `Contact`. Il est préférable d'ajouter une clé étrangère explicite pour garantir que chaque conversation est associée à un contact valide. Il y a d'autres cas où cela est nécessaire, voir la section 4.1.2.

Absence de contraintes `NOT NULL`

Comme déjà expliqué, certaines colonnes dans les tables de la base de données peuvent contenir des valeurs nulles. Cependant, le code Java ne semble pas gérer ces valeurs nulles. Pour éviter des erreurs potentielles, on recommande l'utilisation de contraintes `NOT NULL`.

Celles-ci permettront de garantir la bonne intégrité des données et d'éviter les erreurs potentielles lors d'une évolution du code.

Absence de documentation

Une des premières bonnes pratiques lorsqu'on développe un produit logiciel est de documenter le code. Cela permet de faciliter la compréhension du code par les autres développeurs et de garantir la maintenabilité du code. Cependant, cette application ne contient que très peu de documentation. Il est donc difficile de comprendre le code et de le maintenir.

Notre recommandation serait au mieux de tout redocumenter. Ceci étant dit, on est conscient que cela peut être une tâche très longue et fastidieuse. On recommande donc de documenter au fur et à mesure que l'on modifie le code. Cela permettra de garantir que le code reste documenté et maintenable.

Utilisation de différentes stratégies pour un même problème

Nous avons observé que les stratégies de stockage du `JID`[8] sont différentes entre la table `Contact` et la table `Account`. La table `Contact` opte pour un stockage en une seule colonne, celle-ci de type TEXT. Tandis que la table `Account` utilise trois colonnes, `username`, `server` et `ressource`, pour stocker le `JID`.

Ceci pose un problème de cohérence et de maintenabilité. Il est préférable d'utiliser une seule stratégie pour stocker le `JID`. On recommande l'utilisation d'une seule colonne de type TEXT pour stocker le `JID` dans les deux tables. Et, dans le meilleur des mondes, ajouter un trigger afin de garantir que le `JID` est bien formé.

8. Pour rappel, le `JID` prend cette syntaxe : `username@server/ressource`

Ajout d'une table groupe

Comme vu via les colonnes `Messages.counterpart` ainsi que `Messages.trueCounterpart` ou encore `Conversations.mode`, qui gère les cas où la conversation vient d'un groupe et non d'un message individuel, il serait mieux qu'il y ait une table Groupe qui s'occupe de cette logique et non directement que la feature et la logique soient seulement implémentées dans le code et non via la DB. Actuellement s'il y a un groupe, c'est seulement une instance de la table `Conversations` où le `Conversations.ContactJid` ne référence pas un `Contact` mais un groupe type `Groupe1@unamur.be`.

Évolution du Schéma de la Base de Données

La méthode `onUpgrade()` du fichier `DatabaseBackend.java` est utilisée pour gérer les modifications apportées au schéma de la base de données lors des mises à jour de version. Elle assure la compatibilité ascendante en mettant à jour progressivement la structure et les données pour répondre aux nouveaux besoins de l'application.

Dans cette analyse, nous détaillons les évolutions par version et mettons en lumière les principes clés observés dans ces modifications.

Approche Détournée et Originale

Pour cette analyse, nous avons choisi une méthode détournée et originale : explorer toutes les versions décrites dans `onUpgrade()` pour offrir une vue complète de l'évolution du schéma de base de données. Bien que cela soit une version détournée de la consigne de sélectionner trois versions, cette approche est facilitée par l'utilisation explicite de la méthode `onUpgrade()` et permet d'illustrer chaque étape clé de l'évolution.

Changements par version

Version 1

— **Focus principal :** Création des tables fondamentales `accounts`, `messages`, et `contacts`.
— **Objectif :** Établir les bases minimales pour la gestion des utilisateurs et des messages.

Version 2

— **Modification :** Mise à jour des options dans la table `accounts`.

— **Objectif :** Introduire de nouvelles options pour les comptes utilisateur.

Version 3

— **Modification :** Ajout de la colonne `type` à la table `messages`.
— **Objectif :** Améliorer la classification des messages.

Version 5

— **Modification :** Recréation de la table `contacts` et réinitialisation de la colonne `rosterversion` dans `accounts`.
— **Objectif :** Adapter le stockage des contacts à des besoins évolués.

Version 15

— **Modification :** Introduction des tables liées à OMEMO (`identities`, `prekeys`, etc.) et ajout de la colonne `fingerprint` à `messages`.
— **Objectif :** Renforcer la sécurité par des fonctionnalités cryptographiques.

Version 26

— **Modification :** Création de `presence_templates` et ajout des colonnes `status` et `statusMessage` à `accounts`.
— **Objectif :** Améliorer la gestion des statuts utilisateurs.

Version 39

— **Modification :** Création de la table `resolver_results`.
— **Objectif :** Optimiser le stockage des résultats de résolution DNS.

Version 52

— **Modification :** Ajout des colonnes `occupantId` et `reactions` dans `messages`.
— **Objectif :** Introduire des fonctionnalités interactives comme les réactions.

Analyse des évolutions

— **Compatibilité ascendante :** À travers les versions étudiées, aucune table ou colonne existante n'a été supprimée. Tous les changements se concentrent sur des ajouts progressifs pour enrichir les fonctionnalités tout en garantissant la stabilité des données.
— **Évolution utilitaire :** Les ajouts sont motivés par des besoins spécifiques, tels que la sécurité (version 15) ou l'expérience utilisateur (version 52).
— **Qualité globale :** Ces évolutions démontrent une amélioration continue du système. La sécurité, la performance, et l'expérience utilisateur ont été renforcées sans compromettre la compatibilité ascendante.
— **Vision à long terme :** L'approche incrémentale reflète une architecture pensée pour s'adapter aux futures exigences sans nécessiter de restructuration majeure.

Conclusion

Notre analyse de l'évolution des versions montre une progression maîtrisée et stratégique. Les ajouts successifs, bien qu'incrémentaux, ont significativement contribué à améliorer la qualité globale du système. Cette méthode de migration, combinée à une compatibilité ascendante constante, illustre une gestion efficace et réfléchie de la base de données au sein du projet.