Résultats de l'analyse du schéma et de l'analyse des requêtes (découverte de FK implicites).

1. Relation entre les tables stories et storytext

```
String q = "DELETE FROM " + DatabaseConstants.STORY_TEXT_TABLE +
           " WHERE " + DatabaseConstants.STORY_TEXT_STORY_HASH + " NOT IN " +
           "( SELECT " + DatabaseConstants.STORY_HASH + " FROM " + DatabaseConstants.STORY_TABLE +
           ")";
```

Lors de l'observation du PS nous avons constaté que les noms de colonnes utilisées dans cette requête sont exactement les mêmes (story_hash). Bien que la similarité des noms de colonnes (story_hash) soit un indice sur l'existence d'une clé étrangère, ce n'est pas une garantie. En effet, nous avons vu en cours qu'il existe des cas où des noms de colonnes similaires ne correspondent pas à une relation de clé étrangère donc une analyse approfondie de la requête s'impose.

La condition de suppression utilise une relation entre les tables STORY_TEXT_TABLE(storytext) et STORY_TABLE (stories).

La sous-requête (SELECT " + DatabaseConstants.STORY_HASH + " FROM " + DatabaseConstants.STORY_TABLE + ") renvoie les valeurs de la colonne STORY_HASH (story_hash), qui est une colonne identifiante, de la table STORY_TABLE (stories). Ensuite, dans la requête principale, les lignes de la table STORY_TEXT_TABLE (storytext) sont supprimées si la valeur de la colonne STORY_TEXT_STORY_HASH (story_hash) n'est pas présente dans le résultat de la sous-requête.

La colonne STORY_TEXT_STORY_HASH dans STORY_TEXT_TABLE agit comme une référence vers la colonne STORY_HASH dans STORY_TABLE donc la relation implicite entre ces deux colonnes peut être interprétée comme une clé étrangère implicite.

2. Relation entre les tables stories et feeds

```
String operator = (read ? " - 1" : " + 1");
StringBuilder q = new StringBuilder("UPDATE " + DatabaseConstants.FEED_TABLE);
q.append(" SET ").append(impactedCol).append(" = ").append(impactedCol).append(operator);
q.append(" WHERE " + DatabaseConstants.FEED_ID + " = ").append(story.feedId);
dbRW.execSQL(q.toString());
```

La requête met à jour soit la colonne FEED_NEUTRAL_COUNT soit FEED_POSITIVE_COUNT dans la table FEED_TABLE (Feed).

La justification de l'existence d'une clé étrangère implicite repose sur la relation entre la colonne utilisée dans la condition (FEED_ID (_id) dans FEED_TABLE (Feed)) et la propriété de l'objet story (feedId) qui représente un objet de la table Stories.

Nous pouvons donc supposer qu'il existe une correspondance entre la colonne "_id" dans Feed et la colonne "feed_id" dans la table Stories.

3. Relation entre les tables stories et social_feeds

```
if (!TextUtils.isEmpty(story.socialUserId)) {
    socialIds.add(story.socialUserId);
}

for (String socialId : socialIds) {
    q = new StringBuilder("UPDATE " + DatabaseConstants.SOCIALFEED_TABLE);
    q.append(" SET ").append(impactedSocialCol).append(" = ").append(impactedSocialCol).append(operator);
    q.append(" WHERE " + DatabaseConstants.SOCIAL_FEED_ID + " = ").append(socialId);
    dbRW.execSQL(q.toString());
}
```

La requête met à jour soit la colonne SOCIAL_FEED_NEUTRAL_COUNT soit la colonne SOCIAL_FEED_POSITIVE_COUNT de SOCIALFEED_TABLE en fonction des valeurs de socialIds, qui sont collectées à partir des propriétés de l'objet story.

En regardant le schéma physique nous constatons que les noms des colonnes "socialUserId" dans la table "Stories" et "_id" dans la table "Social_Feeds" ne sont pas explicitement évidents quant à leur rôle de clé étrangère. Cependant, lors de l'examen de la requête nous pouvons établir une relation entre les deux tables en utilisant socialUserId et la colonne identifiante SOCIAL_FEED_ID de la table SOCIALFEED_TABLE

4. Relation entre les tables stories et socialfeed_story_map

```
sel.append("SELECT " + DatabaseConstants.STORY_HASH); [1]


sel.append(" FROM " + DatabaseConstants.SOCIALFEED_STORY_MAP_TABLE);
sel.append(DatabaseConstants.JOIN_STORIES_ON_SOCIALFEED_MAP);
sel.append(" WHERE " + DatabaseConstants.SOCIALFEED_STORY_MAP_TABLE + "." + DatabaseConstants.SOCIALFEED_STORY_USER_ID + " = ? ");
selArgs.add(fs.getSingleSocialFeed().getKey());
DatabaseConstants.appendStorySelection(sel, selArgs, readFilter, stateFilter, fs.getSearchQuery());

public static final String JOIN_STORIES_ON_SOCIALFEED_MAP =
    " INNER JOIN " + STORY_TABLE + " ON " + STORY_TABLE + "." + STORY_ID + " = " + SOCIALFEED_STORY_MAP_TABLE + "." + SOCIALFEED_STORY_STORYID;
```

La requête récupère le story_hash de la table SOCIALFEED_STORY_MAP_TABLE, jointe avec une autre table (STORY_TABLE). La clause "inner join" nous permet de détecter une clé étrangère entre les tables STORY_TABLE et SOCIALFEED_STORY_MAP_TABLE qui se base sur l'égalité entre les colonnes STORY_ID et SOCIALFEED_STORY_STORYID.

5. Relation entre les tables comments et comments_replies

```
public void insertReplyPlaceholder(String storyId, @Nullable String userId, String commentUserId, String replyText) {
    // get a fresh copy of the comment so we can discover the ID
    Cursor c = dbRO.query(DatabaseConstants.COMMENT_TABLE,
                          null,
                          DatabaseConstants.COMMENT_STORYID + " = ? AND " + DatabaseConstants.COMMENT_USERID + " = ?",
                          new String[]{storyId, commentUserId},
                          null, null, null);
    if ((c == null)||(c.getCount() < 1)) {
        com.newsblur.util.Log.w(this, "comment removed before reply could be processed");
        closeQuietly(c);
        return;
    }
    c.moveToFirst();
    Comment comment = Comment.fromCursor(c);
    closeQuietly(c);

    Reply reply = new Reply();
    reply.commentId = comment.id;
    reply.text = replyText;
    reply.userId = userId;
    reply.date = new Date();
    reply.id = Reply.PLACEHOLDER_COMMENT_ID + storyId + comment.id + reply.userId;
    synchronized (RW_MUTEX) {dbRW.insertWithOnConflict(DatabaseConstants.REPLY_TABLE, null, reply.getValues(), SQLiteDatabase.CONFLICT_REPLACE);}
}
```

La première requête récupère un commentaire en fonction d'un storyId et d'un commentUserId. La seconde requête, utilisée pour insérer une réponse dans la table des réponses (REPLY_TABLE), utilise les valeurs de l'objet "reply." L'attribut commentId de l'objet reply a la même valeur que l'identifiant de l'objet "comment"[2], lequel est créé à partir des résultats de la première requête, basée sur la table COMMENT_TABLE.

La colonne commentId dans la table REPLY_TABLE semble être une clé étrangère faisant référence à la clé primaire de la table COMMENT_TABLE.

---

[1] Pour une meilleure visibilité les lignes de codes après cette ligne et la requête ont été supprimées, cfr le dépôt github pour plus de détail sur la méthode.

[2] Plus de détails dans la fonction `Comment fromCursor(final Cursor cursor)` dans la classe Comment.java sur github

6. Relation entre les tables stories et reading_session

```
144⊖ public void cleanupVeryOldStories() {
145      Calendar cutoffDate = Calendar.getInstance();
146      cutoffDate.add(Calendar.MONTH, -1);
147      synchronized (RW_MUTEX) {
148          int count = dbRW.delete(DatabaseConstants.STORY_TABLE,
149                  DatabaseConstants.STORY_TIMESTAMP + " < ?" +
150                  " AND " + DatabaseConstants.STORY_TEXT_STORY_HASH + " NOT IN " +
151                  "( SELECT " + DatabaseConstants.READING_SESSION_STORY_HASH + " FROM " + DatabaseConstants.READING_SESSION_TABLE + ")",
152                  new String[]{Long.toString(cutoffDate.getTime().getTime())});
153          com.newsblur.util.Log.d(this, "cleaned up ancient stories: " + count);
154      }
155  }
```

La méthode supprime les enregistrements de la table STORY_TABLE où la colonne STORY_TIMESTAMP est antérieure à la date de coupure. Nous pouvons constater qu'une requête SELECT est utilisée pour filtrer ceux qui sont référencés dans la table READING_SESSION_TABLE. Cela nous indique qu'il existe une relation entre STORY_TABLE et READING_SESSION_TABLE. Plus précisément une clé étrangère implicite "READING_SESSION_STORY_HASH" qui pointe vers "STORY_TEXT_STORY_HASH" qui est la clé primaire de la table STORY_TABLE

7. Relation entre les tables comments et user_table

Dans la méthode doInBackground[3], les commentaires (comments) sont récupérés à partir de la méthode getComments, qui extrait des commentaires de la table COMMENT_TABLE. Ensuite, pour chaque commentaire, la méthode getUserProfile est utilisée pour obtenir le profil de l'utilisateur, dans la table USER_TABLE, à qui le commentaire est associé. L'outpout de la première requête est donc utilisé comme input de la seconde requête.

Cette utilisation suggère une clé étrangère dans la table COMMENT_TABLE qui fait référence à la clé primaire dans la table USER_TABLE, pour permettre de lier chaque commentaire à un utilisateur spécifique.

```
1329⊖ public List<Comment> getComments(String storyId) {
1330      String[] selArgs = new String[] {storyId};
1331      String selection = DatabaseConstants.COMMENT_STORYID + " = ?";
1332      Cursor c = dbRO.query(DatabaseConstants.COMMENT_TABLE, null, selection, selArgs, null, null, null);
1333      List<Comment> comments = new ArrayList<Comment>(c.getCount());
1334      while (c.moveToNext()) {
1335          comments.add(Comment.fromCursor(c));
1336      }
1337      closeQuietly(c);
1338      return comments;
1339  }

1424⊖ public UserProfile getUserProfile(String userId) {
1425      String[] selArgs = new String[] {userId};
1426      String selection = DatabaseConstants.USER_USERID + " = ?";
1427      Cursor c = dbRO.query(DatabaseConstants.USER_TABLE, null, selection, selArgs, null, null, null);
1428      UserProfile profile = UserProfile.fromCursor(c);
1429      closeQuietly(c);
1430      return profile;
1431  }
```

```
private fun doInBackground() {
    if (context == null || story == null || story.id.isNullOrEmpty()) return
    comments.addAll(fragment.dbHelper.getComments(story.id))

    // users by whom we saw non-pseudo comments
    val commentingUserIds: MutableSet<String> = HashSet()
    // users by whom we saw shares
    val sharingUserIds: MutableSet<String> = HashSet()
    for (comment in comments) {
        // skip public comments if they are disabled
        if (!comment.byFriend && !PrefsUtils.showPublicComments(context)) {
            continue
        }
        val commentUser = fragment.dbHelper.getUserProfile(comment.userId)
```

[3] Lignes 57-70 dans le fichier SetUpCommentSection.kt

8. Relation entre les tables comments_reply et user_table

Dans le code source ci-dessous, les comments_reply sont récupérés à partir de la méthode getCommentsReplies, qui extrait des comments_reply de la table REPLY_TABLE. Pour chaque objet comments_reply, la méthode getUserProfile est utilisée pour obtenir le profil de l'utilisateur, dans la table USER_TABLE. Comme pour l'exemple précédent, l'outpout de la première requête est utilisé comme input de la seconde requête.

Cette utilisation suggère une clé étrangère dans la table REPLY_TABLE qui fait référence à la clé primaire dans la table USER_TABLE, pour permettre de lier chaque comments_reply à un utilisateur spécifique.

```
val replies = fragment.dbHelper.getCommentReplies(comment.id)
for (reply in replies) {
    val replyView = inflater.inflate(R.layout.include_reply, null)
    val replyText = replyView.findViewById<View>(R.id.reply_text) as TextView
    replyText.text = UIUtils.fromHtml(reply.text)
    val replyImage = replyView.findViewById<View>(R.id.reply_user_image) as ShapeableImageView
    val replyUser = fragment.dbHelper.getUserProfile(reply.userId)
```

```
1433  public List<Reply> getCommentReplies(String commentId) {
1434      String[] selArgs = new String[] {commentId};
1435      String selection = DatabaseConstants.REPLY_COMMENTID+ " = ?";
1436      Cursor c = dbRO.query(DatabaseConstants.REPLY_TABLE, null, selection, selArgs, null, null, DatabaseConstants.REPLY_DATE + " ASC");
1437      List<Reply> replies = new ArrayList<Reply>(c.getCount());
1438      while (c.moveToNext()) {
1439          replies.add(Reply.fromCursor(c));
1440      }
1441      closeQuietly(c);
1442      return replies;
1443  }
```

```
1424  public UserProfile getUserProfile(String userId) {
1425      String[] selArgs = new String[] {userId};
1426      String selection = DatabaseConstants.USER_USERID + " = ?";
1427      Cursor c = dbRO.query(DatabaseConstants.USER_TABLE, null, selection, selArgs, null, null, null);
1428      UserProfile profile = UserProfile.fromCursor(c);
1429      closeQuietly(c);
1430      return profile;
1431  }
```

9. Relation entre les tables stories et notify_dismiss

```
363  public static final String STORY_QUERY_BASE_1 =
364      "SELECT " +
365      STORY_COLUMNS +
366      " FROM " + STORY_TABLE +
367      " INNER JOIN " + FEED_TABLE +
368      " ON " + STORY_TABLE + "." + STORY_FEED_ID + " = " + FEED_TABLE + "." + FEED_ID +
369      " WHERE ";
370  public static final String STORY_QUERY_BASE_2 =
371      " GROUP BY " + STORY_HASH;
```

```
381      public static String NOTIFY_FOCUS_STORY_QUERY =
382          STORY_QUERY_BASE_1 +
383          STORY_FEED_ID + " IN (SELECT " + FEED_ID + " FROM " + FEED_TABLE + " WHERE " + FEED_NOTIFICATION_FILTER + " = '" + Feed.NOTIFY_FILTER_FOCUS + "')" +
384          " AND " + STORY_INTELLIGENCE_TOTAL + " > 0 " +
385          STORY_QUERY_BASE_2 +
386          " ORDER BY " + STORY_TIMESTAMP + " DESC";
```

```
1101  public Cursor getNotifyFocusStoriesCursor() {
1102      return rawQuery(DatabaseConstants.NOTIFY_FOCUS_STORY_QUERY, null, null);
1103  }
```

```
967    void pushNotifications() {
968        if (! PrefsUtils.isEnableNotifications(this)) return;
969
970        // don't notify stories until the queue is flushed so they don't churn
971        if (UnreadsService.StoryHashQueue.size() > 0) return;
972        // don't slow down active story loading
973        if (PendingFeed != null) return;
974
975        Cursor cFocus = dbHelper.getNotifyFocusStoriesCursor();
976        Cursor cUnread = dbHelper.getNotifyUnreadStoriesCursor();
977        NotificationUtils.notifyStories(this, cFocus, cUnread, iconCache, dbHelper);
978        closeQuietly(cFocus);
979        closeQuietly(cUnread);
980    }
```

```
38    public static synchronized void notifyStories(Context context, Cursor storiesFocus, Cursor storiesUnread, FileCache iconCache, BlurDatabaseHelper dbHelper) {
39        NotificationManagerCompat nm = NotificationManagerCompat.from(context);
40
41        int count = 0;
42        while (storiesFocus.moveToNext()) {
43            Story story = Story.fromCursor(storiesFocus);
44            if (story.read) {
45                nm.cancel(story.hashCode());
46                continue;
47            }
48            if (dbHelper.isStoryDismissed(story.storyHash)) {
49                nm.cancel(story.hashCode());
50                continue;
51            }
52            if (StoryUtils.hasOldTimestamp(story.timestamp)) {
53                dbHelper.putStoryDismissed(story.storyHash);
54                nm.cancel(story.hashCode());
55                continue;
56            }
```

```
1477    public boolean isStoryDismissed(String storyHash) {
1478        String[] selArgs = new String[] {storyHash};
1479        String selection = DatabaseConstants.NOTIFY_DISMISS_STORY_HASH + " = ?";
1480        Cursor c = dbRO.query(DatabaseConstants.NOTIFY_DISMISS_TABLE, null, selection, selArgs, null, null, null);
1481        boolean result = (c.getCount() > 0);
1482        closeQuietly(c);
1483        return result;
1484    }
```

10. Relation entre les tables folders et folders

```
1611    private Set<String> getFeedIdsRecursive(String folderName) {
1612        Folder folder = getFolder(folderName);
1613        if (folder == null) return emptySet();
1614        Set<String> feedIds = new HashSet<>(folder.feedIds);
1615        for (String child : folder.children) feedIds.addAll(getFeedIdsRecursive(child));
1616        return feedIds;
1617    }
1618 }
```

```
567    public Folder getFolder(String folderName) {
568        String[] selArgs = new String[] {folderName};
569        String selection = DatabaseConstants.FOLDER_NAME + " = ?";
570        Cursor c = dbRO.query(DatabaseConstants.FOLDER_TABLE, null, selection, selArgs, null, null, null);
571        if (c.getCount() < 1) {
572            closeQuietly(c);
573            return null;
574        }
575        Folder folder = Folder.fromCursor(c);
576        closeQuietly(c);
577        return folder;
578    }
```

```
26    public static Folder fromCursor(Cursor c) {
27        if (c.isBeforeFirst()) {
28            c.moveToFirst();
29        }
30        Folder folder = new Folder();
31        folder.name = c.getString(c.getColumnIndex(DatabaseConstants.FOLDER_NAME));
32        folder.parents = DatabaseConstants.unflattenStringList(c.getString(c.getColumnIndex(DatabaseConstants.FOLDER_PARENT_NAMES)
33        folder.children = DatabaseConstants.unflattenStringList(c.getString(c.getColumnIndex(DatabaseConstants.FOLDER_CHILDREN_NA
34        folder.feedIds = DatabaseConstants.unflattenStringList(c.getString(c.getColumnIndex(DatabaseConstants.FOLDER_FEED_IDS)));
35        return folder;
36    }
```

11. Relation entre les tables feed_authors et feeds

La méthode suivante (putFeedAuthorsExtSync) commence par supprimer toutes les entrées dans la table FEED_AUTHORS liées au feedId spécifié. Afin de trouver un lien entre feed_id et l'id de la table Feed nous avons cherché où était appelée cette méthode.

```
1529⊖    private void putFeedAuthorsExtSync(String feedId, Collection<String> authors) {
1530         dbRW.delete(DatabaseConstants.FEED_AUTHORS_TABLE,
1531                     DatabaseConstants.FEED_AUTHORS_FEEDID + " = ?",
1532                     new String[](feedId)
1533                    );
1534         List<ContentValues> valuesList = new ArrayList<ContentValues>(authors.size());
1535         for (String author : authors) {
1536             ContentValues values = new ContentValues();
1537             values.put(DatabaseConstants.FEED_AUTHORS_FEEDID, feedId);
1538             values.put(DatabaseConstants.FEED_AUTHORS_AUTHOR, author);
1539             valuesList.add(values);
1540         }
1541         bulkInsertValuesExtSync(DatabaseConstants.FEED_AUTHORS_TABLE, valuesList);
1542     }
```

Celle-ci est utilisée dans la méthode insertStories[4] et est appelée avec impliedFeedId et feedAuthors comme arguments.

```
421                     putFeedAuthorsExtSync(impliedFeedId, feedAuthors);
```

impliedFeedId est défini comme story.feedId dans une boucle qui itère sur des objets de type Story

```
366                         insertSingleStoryExtSync(story);
367                         // if the story is being fetched for the immediate session, also add the hash to the session table
368                         if (forImmediateReading && story.isStoryVisibleInState(stateFilter)) {
369                             ContentValues sessionHashValues = new ContentValues();
370                             sessionHashValues.put(DatabaseConstants.READING_SESSION_STORY_HASH, story.storyHash);
371                             dbRW.insert(DatabaseConstants.READING_SESSION_TABLE, null, sessionHashValues);
372                         }
373                         impliedFeedId = story.feedId;
374                     }
375                 }
376                 if (apiResponse.story != null) {
377                     if ((apiResponse.story.storyHash == null) || (apiResponse.story.storyHash.length() < 1)) {
378                         com.newsblur.util.Log.e(this, "story received without story hash: " + apiResponse.story.id);
379                         return;
380                     }
381                     insertSingleStoryExtSync(apiResponse.story);
382                     impliedFeedId = apiResponse.story.feedId;
383                 }
```

La colonne feed_id de stories est une référence à la colonne _id de la table feed. Ainsi, feed_id dans FEED_AUTHORS peut être considérée comme une clé étrangère vers la table feed

```
431⊖    private void insertSingleStoryExtSync(Story story) {
432         // pick a thumbnail for the story
433         story.thumbnailUrl = Story.guessStoryThumbnailURL(story);
434         // insert the story data
435         ContentValues values = story.getValues();
436         dbRW.insertWithOnConflict(DatabaseConstants.STORY_TABLE, null, values, SQLiteDatabase.CONFLICT_REPLACE);
```

12. Relation entre les tables feed_tags et feeds

Le même raisonnement que le précédent peut être suivi pour la découverte d'une potentielle clée étrangère entre la table feed_tags et feeds

```
1497⊖    private void putFeedTagsExtSync(String feedId, Collection<String> tags) {
1498         dbRW.delete(DatabaseConstants.FEED_TAGS_TABLE,
1499                     DatabaseConstants.FEED_TAGS_FEEDID + " = ?",
1500                     new String[](feedId)
1501                    );
1502         List<ContentValues> valuesList = new ArrayList<ContentValues>(tags.size());
1503         for (String tag : tags) {
1504             ContentValues values = new ContentValues();
1505             values.put(DatabaseConstants.FEED_TAGS_FEEDID, feedId);
1506             values.put(DatabaseConstants.FEED_TAGS_TAG, tag);
1507             valuesList.add(values);
1508         }
1509         bulkInsertValuesExtSync(DatabaseConstants.FEED_TAGS_TABLE, valuesList);
1510     }
```

[4] Lignes 328-428 dans le fichier BlurDatabaseHelper.java

Celle-ci est utilisée dans la méthode insertStories et est appelée avec impliedFeedId et feedTags comme arguments.

```
410                 putFeedTagsExtSync(impliedFeedId, feedTags);
```