Université de Namur
Faculté des sciences informatiques

Évolution des systèmes : analyse SMOKE DB

GROUPE 3 :
Bardieux Arthur
arthur.barbieux@student.unamur.be
Tang David
david.tang@student.unamur.be
Vandeloise Mikel
mikel.vandeloise@student.unamur.be

Contents

Abstract

Ce document détaille le projet de rétro-ingénierie focalisé sur la base de données de l'application mobile Smoke (version du 27/11/23), développée en Java. Ce travail, mené dans le cadre du cours d'Évolution de systèmes logiciels [INFOM218], vise une analyse approfondie du système de base de données sous-jacent à Smoke, une application de communication pour Android. L'exploration systématique de la base de données a été entreprise, mettant en lumière les schémas physiques, logiques et conceptuels, ainsi que la complexité des requêtes SQL. L'objectif principal de cette étude a été de comprendre en détail les interactions au sein de la base de données, d'évaluer son efficacité et d'identifier les domaines d'amélioration potentielle. L'utilisation d'outils d'analyse comme SQLInspecter et une inspection minutieuse du code source Java ont permis de recueillir des informations précises sur la structure et le fonctionnement de la base de données. Ce processus a éclairé les différents aspects du système ainsi que les zones susceptibles d'optimisation.

1 Etape 1 : Analyse détaillée des accès aux tables et colonnes dans la base de données Smoke

1.1 Méthodologie

Cette analyse s'appuie sur une exploration approfondie des interactions entre le code source Java de l'application Smoke et sa base de données, en mettant une attention particulière sur les requêtes SQL. L'objectif est de décomposer et de comprendre la fréquence, la diversité et la complexité des opérations de base de données. La méthodologie adoptée est articulée autour de plusieurs étapes clés :

1. **Extraction des requêtes SQL :** Utilisation de l'outil SQLInspecter pour extraire systématiquement toutes les requêtes SQL du code source. Cette étape vise à obtenir une vue exhaustive et fidèle des interactions SQL.
2. **Analyse des modèles d'accès :** Évaluation détaillée des modèles d'accès aux données, notamment les tables et colonnes fréquemment consultées. Cette analyse permet d'identifier les motifs récurrents et leur rôle dans l'architecture de l'application.
3. **Étude de la structure des requêtes :** Analyse approfondie de la construction des requêtes SQL pour évaluer leur complexité, y compris les jointures, sous-requêtes et opérations d'agrégation.
4. **Évaluation des implications :** Détermination de l'impact des pratiques actuelles sur la performance, la maintenabilité et la scalabilité, ainsi que l'identification des opportunités d'optimisation.

Cette approche méthodologique offre une compréhension holistique des interactions entre l'application Smoke et sa base de données, formant une base solide pour des recommandations d'optimisation et d'amélioration.

1.2 Résultats

1.2.1 schéma physique

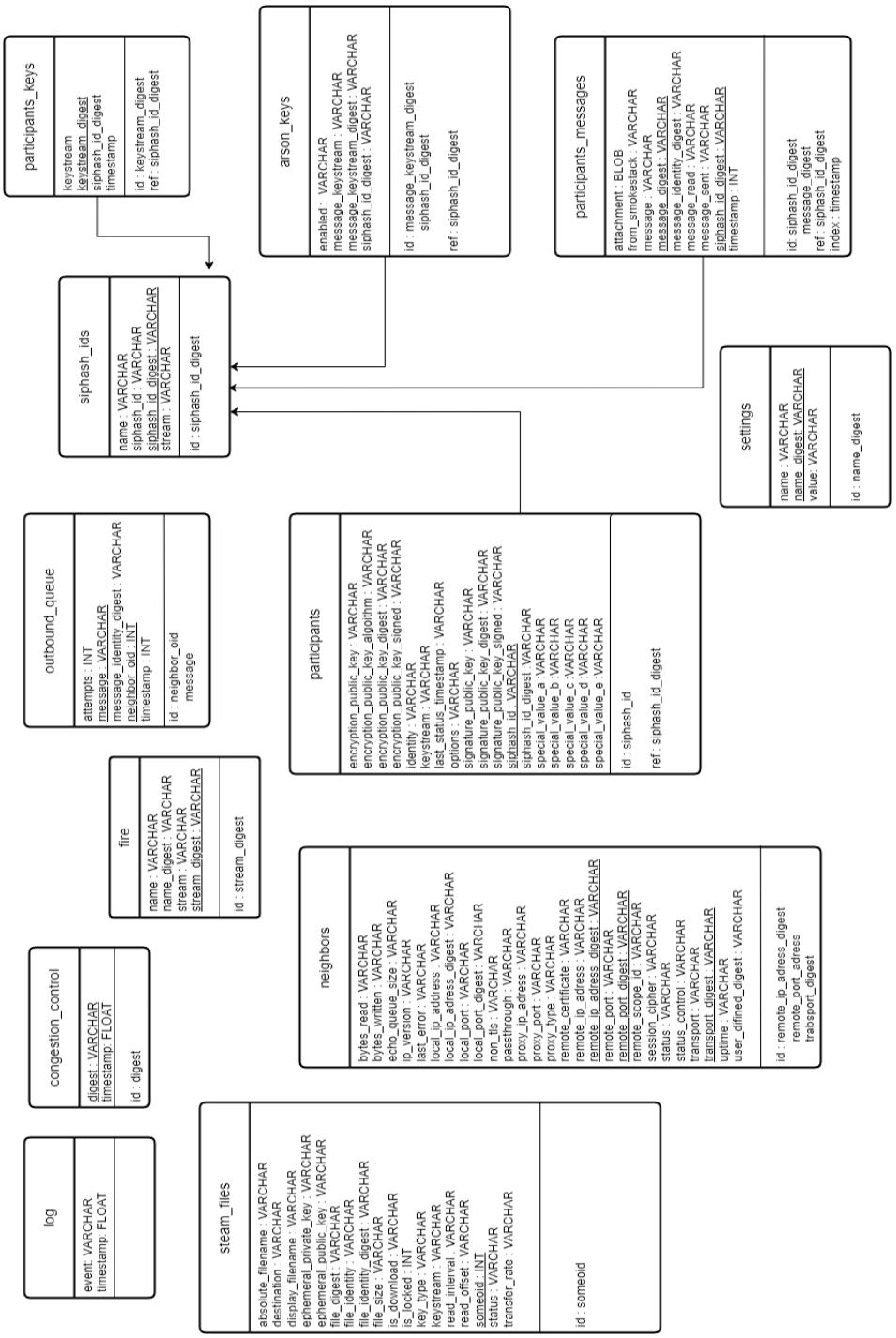

Figure 1: schéma physique

Le schéma physique de la base de données Smoke a été extrait avec SQLInspect dans l'onglet "Schema" et avec les requêtes SQL du fichier Database.java. Ce schéma contient 12 tables et 98 attributs. Les clés primaires et étrangères sont explicitement définies dans les requêtes DDL lors de la création de la table.

Exemple pour la table 'participants_keys':

```
/*
** Create the participants_keys table. Note that the
** keystream_digest should be unique for all participants.
*/

str = "CREATE TABLE IF NOT EXISTS participants_keys (" +
    "keystream TEXT NOT NULL, " + /*
                  ** Authentication and encryption
                  ** keys.
                  */
    "keystream_digest TEXT NOT NULL PRIMARY KEY, " +
    "siphash_id_digest TEXT NOT NULL, " +
    "timestamp DATETIME DEFAULT CURRENT_TIMESTAMP, " +
    "FOREIGN KEY (siphash_id_digest) REFERENCES " +
    "siphash_ids (siphash_id_digest) ON DELETE CASCADE)";
```

Figure 2: create partipants_keys table

Il a été observé que chaque table de la base de données possède une clé primaire, à l'exception notable de la table '`log`'. De plus, une particularité remarquée dans quatre tables spécifiques, à savoir '`outbound_queue`', '`arsons_keys`', '`participants_messages`' et '`neighbors`', est la présence d'identifiants composites. Ces identifiants sont caractérisés par au moins deux colonnes, témoignant d'une conception structurée pour une gestion de données complexe.

Concernant les clés étrangères, un modèle cohérent a été identifié. La colonne '`siphash_id_digest`' sert de clé étrangère dans les tables '`participants_keys`', '`arson_keys`', '`participants_messages`' et '`participants`', établissant toute une relation référentielle avec la table '`siphash_ids`'. Cette uniformité dans l'utilisation des clés étrangères reflète une intégration soignée des différentes tables et souligne la relation interconnectée entre elles.

1.2.2 schéma logique

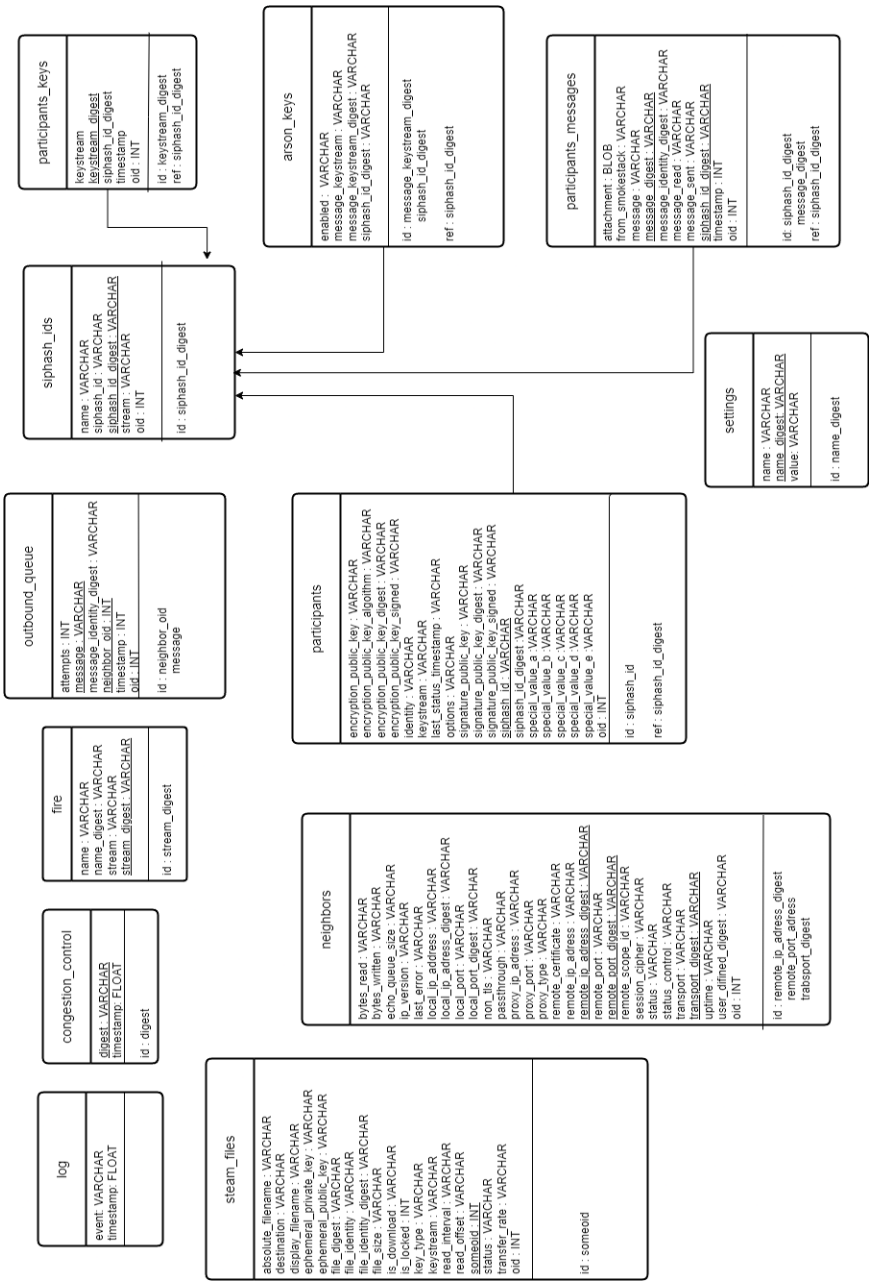

Figure 3: schéma logique

En examinant minutieusement les requêtes SQL présentes dans le fichier `Database.java`, une particularité a été relevée concernant certaines requêtes `select`. Il a été observé que l'attribut `oid` apparaît de manière non évidente dans plusieurs tables : `outbound_queue`, `siphash_ids`, `participants_keys`, `participants_messages`, `participants`, `neighbors`, et `steam_files`. En réponse à cette découverte, la colonne `oid` a été intégrée dans le schéma logique pour refléter cette utilisation cachée. Cette modification a mis en évidence une similitude notable entre les schémas physique et logique, soulignant la cohérence structurelle entre ces deux niveaux de représentation de la base de données. Sur base des analyses des requêtes, aucune clé étrangère implicite n'a été découverte. Pour mieux correspondre au schéma logique, l'index `timestamp` de la table `participants_messages` a été retiré du schéma.

1.2.3 schéma conceptuel

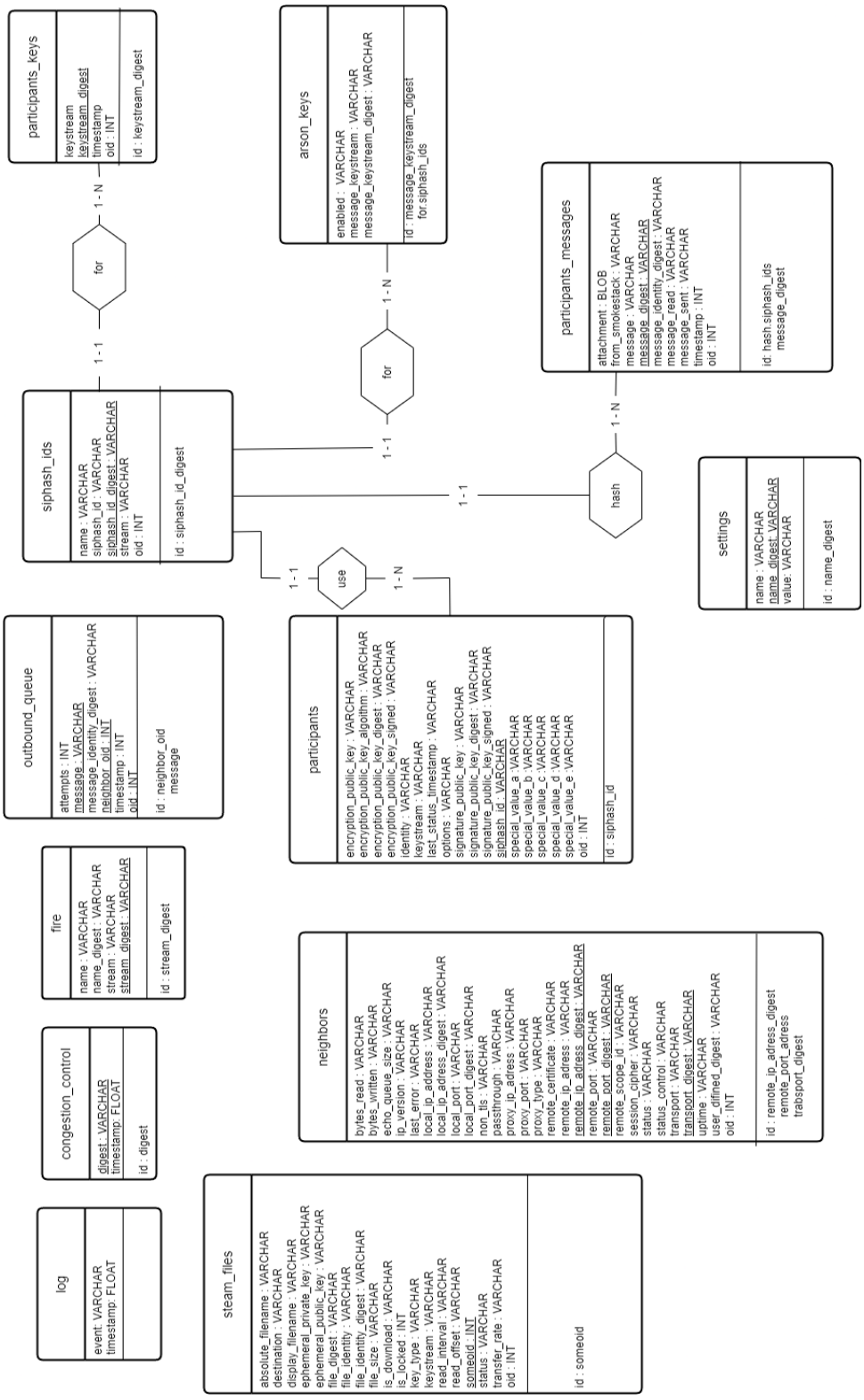

Figure 4: schéma conceptuel

La majorité des clés étrangères '`siphash_id_digest`' ont été converties en relations simples entre entités, adoptant généralement une cardinalité de type un-à-plusieurs. Un cas particulier a été relevé dans les tables '`arson_keys`' et '`participants_messages`', où les clés étrangères fonctionnent également comme des clés primaires, caractérisées par une superposition des identifiants et des clés étrangères (overlapping identifier-foreign key). Dans le schéma conceptuel, ces clés primaires sont adaptées pour refléter cette particularité. Par exemple, dans la table '`participants_messages`', la clé primaire est définie comme `hash.siphash_ids`, soulignant une approche de modélisation de données unique et intégrée.

1.3 Analyse structurale des tables et des requêtes dans la base de données Java du projet Smoke

1.3.1 Méthodologie

L'analyse approfondie des données JSON, représentant la structure de la base de données Java du projet Smoke, a permis d'examiner les identifiants uniques, les relations entre les tables et les requêtes SQL. Cette approche méthodologique inclut l'analyse de la complexité des requêtes, l'évaluation de leur performance et l'identification des opportunités d'optimisation. Des outils analytiques ont été utilisés tels que "AndroidHotspotFinder" pour repérer les requêtes les plus sollicitées et potentiellement problématiques.

1.3.2 Architecture des tables et des requêtes

Utilisation prédominante de certaines tables et colonnes : L'analyse révèle que des tables telles que '`participants`', '`siphash_ids`', '`steam_files`', et '`neighbors`' jouent un rôle fondamental dans les opérations de la base de données. La table '`neighbors`', par exemple, est fréquemment sollicitée pour des fonctions critiques telles que le contrôle d'état et la gestion des connexions réseau, indiquant son importance stratégique. Similairement, '`steam_files`' est essentielle pour la gestion avancée des fichiers et la manipulation de données, révélant une utilisation complexe et variée.

Complexité et sophistication des requêtes : L'examen des requêtes SQL met en évidence une prédominance de commandes avancées, caractérisées par l'intégration de multiples tables et un grand nombre de colonnes. Cette complexité est manifeste dans les requêtes multi-tables, impliquant des jointures élaborées entre des tables comme '`participants`' et '`siphash_ids`'. Ces requêtes indiquent une interaction multidimensionnelle avec la base de données, soulignant une architecture complexe et la nécessité d'une gestion optimisée des données.

1.4 Analyse et recommandations

1.4.1 Optimisation et performance

La performance des requêtes, en particulier celles avec des jointures complexes et des opérations de tri étendues, nécessite une attention particulière. L'amélioration de l'indexation, l'application de stratégies de pagination, et la limitation des résultats retournés sont des recommandations clés pour accroître l'efficacité.

1.4.2 Maintenance et évolutivité

La structure complexe des requêtes et la croissance attendue des données exigent une planification proactive. Les stratégies de partitionnement des tables et l'adoption de modèles de conception de base de données évolutifs sont cruciales pour gérer efficacement de grands ensembles de données et faciliter les mises à niveau.

1.4.3 Sécurité et confidentialité des données

La gestion sécurisée des clés et des identifiants uniques est essentielle. Renforcer la gestion des clés, appliquer des protocoles de cryptage robustes, et mettre en place des politiques de sécurité strictes sont recommandés. Une revue régulière des structures de requêtes aidera également à identifier et à atténuer les vulnérabilités.

1.4.4 Discussion et implications générales

L'analyse souligne l'importance de l'optimisation des performances, de la maintenance proactive et de la sécurité renforcée. Ces domaines clés garantiront l'intégrité, la performance, et la sécurité des données au fil du temps, en particulier dans la manipulation de données confidentielles.

2 Étape 2: Analyse des requêtes et sous-schéma logique

2.1 Méthodologie

Cette section est dédiée à une analyse poussée des requêtes SQL, fondée sur les données extraites du code source Java de l'application. Elle se concentre spécifiquement sur l'identification des modèles d'accès aux données, l'examen de la complexité intrinsèque des requêtes, et l'évaluation de leur impact sur les performances globales de la base de données. Cette approche méthodologique permet de dégager des insights précis sur la manière dont les requêtes SQL interagissent avec la base de données, et de déterminer les potentiels goulots d'étranglement ainsi que les opportunités d'optimisation.

2.2 Analyse des requêtes

Une analyse minutieuse des requêtes SQL a été effectuée pour évaluer l'utilisation et la fréquence d'accès aux différentes tables et colonnes au sein de l'application. Cette étude approfondie a révélé le rôle stratégique de certaines tables clés, ainsi que les relations complexes et interdépendantes qui les lient. L'accent a été mis sur la compréhension de la façon dont ces éléments interagissent au sein de la base de données, fournissant ainsi une vue d'ensemble essentielle sur la dynamique et la structure des opérations de données.

2.2.1 Exemples spécifiques de requêtes

- **Table Neighbors** : Des requêtes fréquentes sur cette table montrent une utilisation intensive pour la gestion des connexions réseau. Elles impliquent des sous-requêtes imbriquées, ce qui indique une complexité significative dans les opérations de réseau.
- **Tables Participants et Participants_Keys** : Ces tables sont souvent interrogées conjointement à travers des jointures complexes pour récupérer et encadrer des informations sensibles liées aux utilisateurs. Cela souligne leur rôle central dans la sécurité et la gestion des identités.

2.2.2 Statistiques

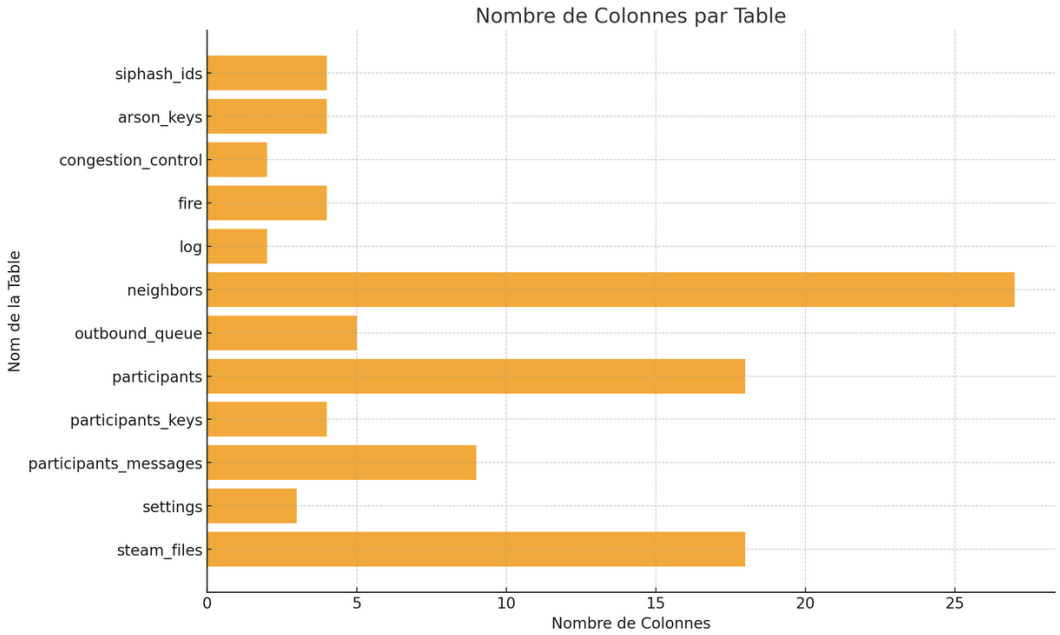

Figure 5: nombre de colonnes par tables

Analyse des attributs par table (Figure 5) La Figure 5 met en évidence que les tables `neighbors`, `participants`, `steam_files`, et `participants_messages` contiennent un nombre élevé d'attributs. Cette représentation montre explicitement les tables les plus riches en termes de quantité d'attributs, indiquant potentiellement une complexité plus grande dans leur gestion et utilisation.

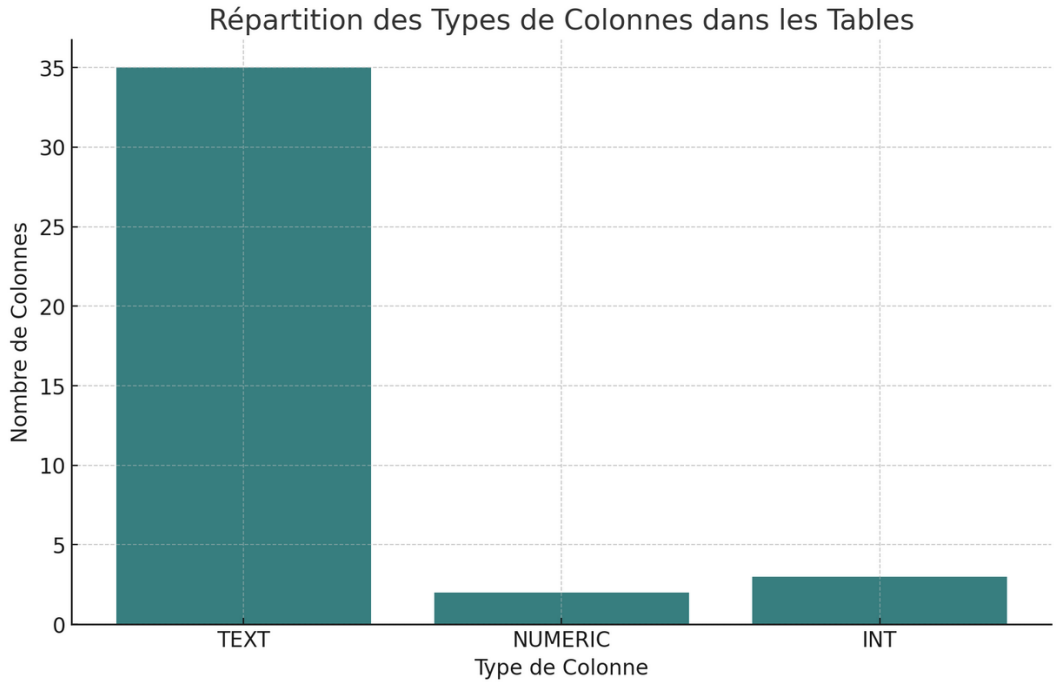

Figure 6: Répartition des types de colonnes dans les tables

Répartition des types de données (Figure 6) La Figure 6 révèle la diversité des types de données utilisés dans la base de données, avec une prédominance des types "TEXT", "NUMERIC", et "INT". Il est également important de noter la présence du type "BLOB" dans la colonne `attachment` de la table `participants_messages`, qui, bien que représentant un seul attribut, est essentiel pour comprendre la gestion des données complexes.

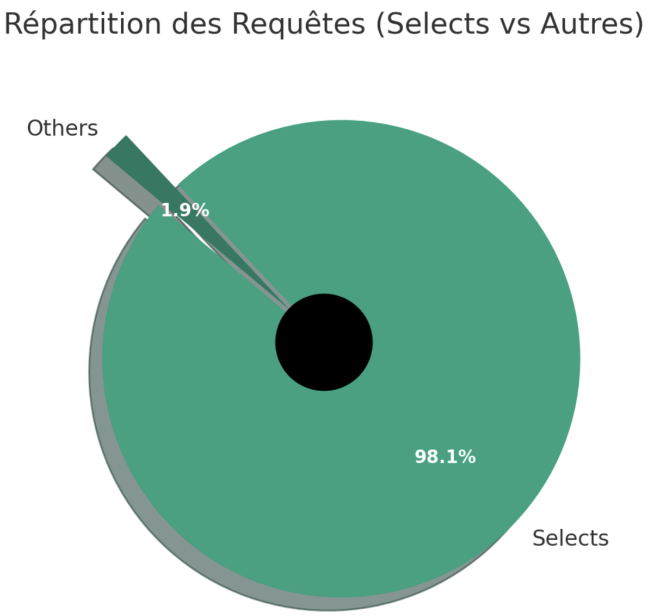

Figure 7: répartition des requêtes

Distribution des types de requêtes SQL (Figure 7) Selon la Figure 7, une majorité écrasante des requêtes SQL effectuées dans l'application sont des requêtes de type "select", représentant plus de 98% du total. Cela indique une utilisation intensive de requêtes de lecture par rapport à d'autres opérations telles que les insertions, mises à jour ou suppressions, qui ne constituent ensemble que 2% des requêtes.

2.3 Dérivation du sous-schéma logique (LSS)

L'analyse détaillée des requêtes a aidé à affiner le LSS de l'application. Évidemment, les tables `participants`, `neighbors`, `siphash_ids`, et `steam_files` sont essentielles, tandis que d'autres tables montrent une utilisation moindre. Cette hiérarchisation du schéma reflète directement les priorités et les fonctions clés de l'application.

2.4 Implications et recommandations

- **Optimisation et Réorganisation des Requêtes** : Pour améliorer les performances, une révision des requêtes complexes est recommandée. L'optimisation peut inclure la réécriture des sous-requêtes, l'utilisation de vues matérialisées, et l'ajustement des jointures pour réduire la charge sur la base de données.
- **Indexation et performance** : Une attention particulière doit être portée à l'indexation des colonnes fréquemment utilisées. Cela accélérera les temps de réponse, surtout pour les requêtes complexes s'appuyant sur des jointures multiples.

- **Sécurité et Intégrité des Données** : Les requêtes mettant en œuvre des données sensibles, telles que les identifiants des utilisateurs, nécessitent des mesures de sécurité accrues. L'implémentation de protocoles de cryptage robustes et la sécurisation des requêtes sont essentielles pour maintenir l'intégrité et la confidentialité des données.

2.5 Conclusion de l'étape 2

L'analyse minutieuse des requêtes SQL a permis de dresser un portrait précis de l'utilisation de la base de données dans l'application. Cette étape révèle non seulement les tables et les requêtes critiques pour les opérations de l'application, mais offre aussi des pistes concrètes pour améliorer les performances et de la sécurité. Les résultats de cette étape constituent un fondement solide pour les optimisations futures et la maintenance du système.

3 Étape 3: Analyse d'impact des scénarios d'évolution du sous-schéma physique

3.1 Méthodologie

Cette étape implique une analyse détaillée des impacts de dix scénarios d'évolution du sous-schéma physique (PSS) sur le système. L'accent est mis sur les modifications requises dans le code source pour maintenir la cohérence globale.

3.2 Scénarios d'évolution et leurs impacts sur le programme

Chaque scénario envisagé est analysé en termes de changements nécessaires dans les classes, méthodes et lignes de code spécifiques.

3.2.1 Détail des scénarios et des impacts correspondants

1. **Renommer des colonnes dans 'neighbors'** :
   - *Classe concrète affectée* : Database.java.
   - *Fonctionnalités affectées* : NetworkManager, ConnectionHandler.
   - *Colonnes renommées* : 'ip_version' en 'ip_proto', 'status_control' en 'status_flag'.
   - *Modifications* : Mise à jour des noms de colonnes dans les requêtes SQL de NetworkManager et ConnectionHandler, ajustement des méthodes de gestion des connexions pour refléter ces changements.
2. **Ajout de colonnes de sécurité dans 'participants'** :
   - *Classe concrète affectée* : Database.java.
   - *Fonctionnalités affectées* : UserAuthenticator, SecurityManager.
   - *Nouvelles colonnes* : 'security_level', 'last_verified'.
   - *Modifications* : Extension des méthodes d'authentification dans UserAuthenticator pour inclure les vérifications de 'security_level', mise à jour des scripts de validation dans SecurityManager pour gérer 'last_verified'.
3. **Suppression de colonnes dans 'siphash_ids'** :
   - *Classe concrète affectée* : Database.java.

- *Fonctionnalités affectées* : DataProcessor, SipHashManager.
- *Colonnes supprimées* : ‘siphash_id_digest‘.
- *Modifications* : Suppression des références à ‘siphash_id_digest‘ dans DataProcessor, révision des algorithmes de traitement de données dans SipHashManager pour s’adapter à cette suppression.

4. **Division de la table ‘steam_files‘** :
   - *Classe concrète affectée* : Database.java.
   - *Fonctionnalités affectées* : FileManager, StreamHandler.
   - *Nouvelles tables* : ‘stream_files‘, ‘file_metadata‘.
   - *Modifications* : Séparation des méthodes de gestion des fichiers dans FileManager, adaptation aux nouvelles structures de tables dans StreamHandler.
5. **Fusion de ‘log‘ et ‘events‘** :
   - *Classe concrète affectée* : Database.java.
   - *Fonctionnalités affectées* : EventLogger, SystemMonitor.
   - *Table fusionnée* : ‘log_events‘.
   - *Modifications* : Intégration des fonctionnalités de logging et d’événements dans EventLogger, consolidation des requêtes dans SystemMonitor pour interagir avec la nouvelle table ‘log_events‘.
6. **Modification des types de données dans ‘settings‘** :
   - *Classe concrète affectée* : Database.java.
   - *Fonctionnalités affectées* : ConfigManager, SettingsValidator.
   - *Modifications de type* : ‘value‘ de TEXT à JSON.
   - *Modifications* : Mise à jour des validateurs de données dans SettingsValidator pour le nouveau type JSON, adaptation des méthodes de lecture et écriture de configuration dans ConfigManager.
7. **Introduction d’une table d’association** :
   - *Classe concrète affectée* : Database.java.
   - *Fonctionnalités affectées* : RelationManager, DatabaseConnector.
   - *Nouvelle table* : ‘users_groups‘.
   - *Modifications* : Création de nouvelles méthodes dans RelationManager pour gérer les relations many-to-many entre ‘users‘ et ‘groups‘, ajustement des requêtes dans DatabaseConnector.
8. **Changement de stratégie d’Indexation sur ‘participants_keys‘** :
   - *Classe concrète affectée* : Database.java.
   - *Fonctionnalités affectées* : IndexOptimizer, QueryBuilder.
   - *Modifications d’indexation* : Ajout d’index sur ‘keystream‘, ‘timestamp‘.
   - *Modifications* : Révision des stratégies d’indexation dans IndexOptimizer, optimisation des requêtes pour améliorer les performances dans QueryBuilder.
9. **Restructuration des clés primaires et étrangères** :

- *Classe concrète affectée* : Database.java.
- *Fonctionnalités affectées* : ModelUpdater, SchemaValidator.
- *Modifications des clés* : Changement de clés primaires dans 'participants', ajout de clés étrangères dans 'participants_messages'.
- *Modifications* : Réécriture des définitions de clés dans ModelUpdater, mise à jour des contraintes et des relations dans le modèle de données.

10. **Optimisation des requêtes pour 'neighbors'** :
    - *Classe concrète affectée* : Database.java.
    - *Fonctionnalités affectées* : QueryOptimizer, NetworkQueryHandler.
    - *Modifications de requêtes* : Optimisation des requêtes pour la sélection de 'status', 'uptime'.
    - *Modifications* : Révision et optimisation des requêtes dans QueryOptimizer, adaptation des méthodes de gestion des requêtes réseau dans NetworkQueryHandler.

3.2.2 Implications des scénarios

- **Maintenance et adaptation** : Les scénarios révèlent des besoins variés en termes de mise à jour du code pour maintenir l'intégrité et l'efficacité.
- **Performance et intégrité** : Les modifications proposées pourraient significativement affecter la performance des requêtes et l'intégrité des données.
- **Sécurité des données** : Certains scénarios soulignent la nécessité de renforcer la sécurité, notamment lors de la manipulation des schémas liés aux informations sensibles.

3.3 Conclusion de l'étape 3

Cette analyse approfondie "what-if" fournit une compréhension claire des répercussions des changements de schéma sur le code source. Elle met en évidence les domaines nécessitants des ajustements pour assurer la cohérence et l'efficacité du système face à divers scénarios d'évolution. Les résultats de cette étape sont essentiels pour guider les décisions de développement et de maintenance futures.

4 Étape 4: Évaluation de la qualité et recommandations

4.1 Objectif de l'étape

Cette étape consiste à analyser en détail la qualité du schéma de la base de données et du code de manipulation de la base de données de l'application Smoke. L'objectif est d'identifier les points forts, les domaines à améliorer et de formuler des recommandations spécifiques pour optimiser le système.

4.2 Évaluation de la qualité du schéma de la base de données

4.2.1 Points forts

- **Cohérence et normalisation** : Analyse des fichiers 'Settings-schema.json' et 'Settings-queries.json' révèle une structure de données simple et bien organisée, avec une utilisation efficace des clés primaires et étrangères pour assurer l'intégrité relationnelle. Un nombre de tables suffisant et des relations entre les tables peu complexes.
- **Clarté des nomenclatures** : Les noms des tables et des colonnes en général sont explicitement définis, reflétant leur fonction et facilitant la compréhension du schéma.

4.2.2 Points faibles

- **Peu d'index** : Dans les schémas, il existe seulement un index dans la table '`participants_messages`' qui permet des accès plus rapidement aux données de la table.
- **Manque de clarté sur des attributs** : Tous les attributs "`special_value_...`" de la table '`participants`' ne sont pas clairs, ce qui diminue leurs compréhensions.

4.2.3 Axes d'amélioration

- **Optimisation des index** : Une analyse approfondie des requêtes SQL montre que l'optimisation des index sur les colonnes fréquemment interrogées dans ces tables pourrait significativement améliorer les performances.
- **Renommage des attributs** : Les noms attributs '`special_value_a`', '`special_value_b`', '`special_value_c`', '`special_value_d`', '`special_value_e`' de la table '`participants`' devraient être renommés, par exemple, en 'phone_number' car ils ne sont pas très explicites.

4.3 Évaluation du code de manipulation de la base de données

4.3.1 Points forts

- **Structuration et organisation** : Le code, comme observé dans les fichiers 'Settings-taa.xml' et 'Settings-smells.xml', présente une structuration logique et une organisation cohérente qui facilitent la maintenance.
- **Isolation de la classe Database** : Les manipulations de la base de données sous forme de requêtes SQL se font tous dans le fichier Database.java. Cela facilite la maintenance en cas de modifications à la base de données.
- **Gestion des exceptions** : Une gestion des erreurs et des exceptions dans les interactions avec la base de données est observée, ce qui garantit la robustesse de l'application.

4.3.2 Points faibles

- **Requêtes complexes** : Certaines requêtes contiennent des requêtes imbriquées. Cela peut avoir un impact sur les performances et augmenter la charge sur le SGBD si les sous-requêtes renvoient un grand résultat. De plus, cette complexité rend le code moins lisible et moins compréhensible pour sa maintenance.
- **Vulnérabilité aux injections SQL** : La présence de requêtes non paramétrées dans le programme constitue une faille de sécurité majeure, ouvrant la porte à des attaques par injection SQL. Ces attaques exploitent les requêtes mal sécurisées pour manipuler ou accéder de manière non autorisée à la base de données, ce qui peut entraîner la compromission des données sensibles, leur altération ou leur suppression.

4.3.3 Axes d'amélioration

- **Optimisation des requêtes SQL** : Les requêtes complexes identifiées dans 'Settings-queries.json' nécessitent une optimisation pour réduire le temps de réponse et la charge sur le serveur de base de données.
- **Sécurité des requêtes** : Bien que la gestion des exceptions soit robuste, des améliorations dans la prévention des injections SQL et dans la sécurisation des données sensibles sont nécessaires.

4.4 Recommandations pour l'amélioration

- **Révision de l'indexation** : Réaliser une analyse détaillée des plans d'exécution des requêtes pour identifier les opportunités d'indexation qui pourraient améliorer les performances.
- **Optimisation des requêtes SQL** : Revoir les requêtes complexes, en particulier celles impliquant des jointures multiples ou des sous-requêtes, pour les rendre plus efficaces.
- **Renforcement de la sécurité** : Mettre en œuvre des mesures de prévention contre les injections SQL, comme l'utilisation de requêtes paramétrées et le renforcement de la validation des entrées utilisateur.

4.5 Conclusion de l'étape 4

L'analyse détaillée fournie dans cette étape met en évidence des aspects cruciaux de la qualité du schéma, de la base de données et du code de manipulation. Les recommandations proposent des pistes concrètes pour améliorer la performance, la sécurité et la facilité de maintenance, contribuant ainsi à une évolution positive et durable de l'application Smoke.

5 Bonus

5.1 Complexité croissante et implications sur la performance

L'évolution des requêtes SQL dans "Smoke" durant trois dernières années montre une tendance vers une complexité accrue. Initialement composée de requêtes simples, comme *"SELECT name, oid FROM fire"*, l'application a progressivement adopté des requêtes plus élaborées avec sous-requêtes et jointures. Bien que cette évolution témoigne d'une fonctionnalité enrichie, elle soulève des préoccupations concernant la performance et la maintenabilité. Par exemple, des requêtes telles que *"SELECT... FROM neighbors n ORDER BY n.oid"* nécessitent une optimisation minutieuse pour prévenir les goulets d'étranglement.

5.2 Sécurité et confidentialité : progression et défis

Au fil des versions, "Smoke" a intégré des fonctionnalités de cryptographie avancées, y compris l'usage de *Base64.encodeToString* et *cryptography.hmac*. Cependant, cette progression introduit également des risques liés à une utilisation inappropriée de la cryptographie et à une gestion des clés insuffisante, pouvant compromettre la sécurité globale du système.

5.3 Optimisation des performances : un processus evolutif

L'utilisation de clauses comme *WHERE*, *ORDER BY*, et *LIMIT* dans les requêtes telles que *"SELECT... FROM steam_files WHERE ... ORDER BY someoid LIMIT 1"* indique des efforts d'optimisation des performances. Néanmoins, ce processus d'optimisation doit être continu et adaptatif pour répondre aux exigences changeantes des versions successives de l'application.

5.4 Adaptabilité et évolutivité : enjeux de conception

La diversité croissante des requêtes SQL au fil des ans reflète l'adaptabilité de "Smoke" aux besoins changeants. Cependant, cette adaptabilité peut aussi indiquer des défis en matière de planification à long terme et de cohérence dans la conception de la base de données, affectant potentiellement la stabilité future du système.

5.5 Cohérence et maintenance : nécessité d'une stratégie unifiée

Bien que l'ajout régulier de nouvelles requêtes soit nécessaire pour répondre aux besoins évolutifs, il est impératif de maintenir des normes de développement cohérentes et une stratégie de maintenance claire. Les modifications dans les requêtes au fil des versions doivent être guidées par des principes de conception uniformes pour assurer la stabilité et la fiabilité à long terme.

5.6 Conclusion de l'analyse sur trois ans

Cette analyse triennale des requêtes SQL de "Smoke" révèle un développement fonctionnel progressif, mais souligne également des défis significatifs en termes de performance, de sécurité, d'adaptabilité et de maintenance. Il est crucial d'adopter une approche équilibrée et une analyse rigoureuse pour évaluer et améliorer la gestion des données de l'application durablement.

6 Source

Github du groupe 3: `https://github.com/David5031/Evolution-systemes-logiciels`
Github de l'application Smoke: `https://github.com/textbrowser/smoke`