MSE | MASTER OF SCIENCE IN ENGINEERING

Hes·so
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences and Arts Western Switzerland

Master of Science HES-SO in Engineering
Av. de Provence 6
CH-1007 Lausanne

# Master of Science HES-SO in Engineering

## Technologies de l'information et de la communication (TIC)

# Migration d'une base de données relationnelle en une base de données orientée événements : comment gérer la cohérence des données, les transactions et l'indexation.

Fait par

Steven Liatti

Sous la direction de
Prof. Paul Albuquerque
Joël Cavat
HEPIA

Expert externe Prof. Florent Glück

Genève, HES-SO//Master, 2021

Accepté par la HES-SO//Master (Suisse, Lausanne) sur proposition de Prof. Paul Albuquerque

Prof. Paul Albuquerque, conseiller du projet d'approfondissement
Florent Glück, Expert principal

Lausanne, le 4 juin 2021

Prof. Paul Albuquerque
Conseiller

Prof. Nabil Abdennadher
Responsable de la filière

# Table des matières

Table des matières . . . 3
Table des figures . . . 5
Table des tables . . . 5
Table des listings de code source . . . 6
Conventions typographiques . . . 7
Structure du document . . . 7
Remerciements . . . 7
Acronymes . . . 8

**1 Introduction 10**
1.1 Motivations . . . 10
1.2 Buts . . . 10

**2 Analyse architecturale et technologique 11**
2.1 Les bases de données relationnelles et MySQL . . . 11
2.2 *Change Data Capture* . . . 13
2.3 *Event Sourcing* . . . 13
2.4 CQRS *pattern* . . . 15
2.5 Le *message broker* Kafka . . . 16
2.5.1 Messages et persistance . . . 17
2.5.2 Topics . . . 17
2.5.3 Producteurs . . . 18
2.5.4 Consommateurs . . . 19
2.5.5 Partitions . . . 20
2.5.6 Streams . . . 20
2.6 Alternatives à Kafka . . . 21
2.6.1 ActiveMQ . . . 21
2.6.2 RabbitMQ . . . 22
2.6.3 Logstash et suite ELK . . . 22
2.6.4 Choix final porté sur Kafka . . . 22
2.7 Debezium . . . 23
2.8 ksqlDB . . . 25

**3 Implémentations 29**
3.1 Modèle initial, génération de données et base commune . . . 29
3.1.1 Modèle initial . . . 29
3.1.2 Génération de données . . . 30
3.1.3 Base commune . . . 31

3.2 Purement relationnelle . . . 33
3.2.1 Concept . . . 33
3.2.2 Code . . . 36
3.3 Orienté message broker avec Kafka . . . 40
3.3.1 Concept . . . 40
3.3.2 Code . . . 40
3.4 MySQL + Debezium : changements kafkaïens . . . 44
3.4.1 Concept . . . 44
3.4.2 Code . . . 45
3.5 ksqlDB : historique avec *streams* et vues matérialisées . . . 47
3.6 MySQL + Debezium + Kafka Streams . . . 50

**4 Discussions et comparaisons des implémentations 53**
4.1 Purement relationnelle . . . 53
4.1.1 Avantages . . . 53
4.1.2 Inconvénients . . . 53
4.2 Kafka uniquement . . . 53
4.2.1 Avantages . . . 53
4.2.2 Inconvénients . . . 54
4.3 MySQL + Debezium . . . 54
4.3.1 Avantages . . . 54
4.3.2 Inconvénients . . . 55
4.4 ksqlDB only . . . 55
4.4.1 Avantages . . . 55
4.4.2 Inconvénients . . . 55
4.5 MySQL + Debezium + Kafka Streams . . . 55
4.5.1 Avantages . . . 55
4.5.2 Inconvénients . . . 56
4.6 Récapitulatif et synthèse . . . 56

**5 Conclusion 58**
5.1 Bilan . . . 58
5.2 Problèmes rencontrés . . . 58
5.3 Améliorations possibles . . . 58

**6 Références 59**

# Table des figures

1 Séparation de l'information entre plusieurs tables - Joël Cavat [1] . . . . . 11
2 Schéma de principe du *pattern* CQRS . . . . . 15
3 Illustration de la répartition des messages dans différentes partitions - Kafka documentation [20] . . . . . 20
4 Architecture et proposition de pipeline de Debezium - Debezium [34] . . . . . 23
5 Confrontation entre bases de données relationnelles et ksqlDB sur les requêtes et données - Confluent Inc. [40] . . . . . 26
6 Pipeline de traitement de données *ante* ksqlDB - Confluent Inc. [40] . . . . . 28
7 Pipeline de traitement de données avec ksqlDB - Confluent Inc. [40] . . . . . 28
8 Diagramme entité-associations initial . . . . . 29
9 Diagramme relationnel initial . . . . . 30
10 Table `Client` avec champ `timestamp` supplémentaire . . . . . 34
11 Diagramme relationnel avec *tracking* des changements . . . . . 35
12 Spectre des implémentations, triées en fonction du *pattern* . . . . . 57

# Table des tables

1 Exemple d'une séquence de transactions d'un compte bancaire . . . . . 14
2 Comparaison des effets de chaque opération CRUD sur les *streams* et les tables - Michael Noll [38] . . . . . 26
3 Exemple de mises à jour d'un client . . . . . 34

# Table des listings de code source


1 Hello world en Scala, exemple d'un extrait de code . . . 7
2 Exemple de message Kafka, au format JSON . . . 17
3 Exemple de producteur Kafka en Scala - Loïc Divad [19] . . . 19
4 Exemple de consommateur Kafka en Scala . . . 19
5 Exemple de *stream* Kafka en Scala - Kafka Streams Documentation [21] . . . 21
6 Exemple d'une clé d'un événement publié dans Kafka depuis Debezium, au format JSON . . . 24
7 Exemple d'une valeur d'un événement publié dans Kafka depuis Debezium, au format JSON . . . 25
8 Exemple de requête *push* ksqlDB . . . 27
9 Extrait du trait `FlyManager` . . . 32
10 Différences entre le type `Client` et `ClientHistory` . . . 32
11 Enumération `Status` . . . 33
12 Les trois fonctions de commandes et requêtes de `MySQLFlyManager` . . . 37
13 Quelques fonctions basiques de `MySQLFullFlyManager` . . . 37
14 Fonction `updateBookingSeat` nécessitant l'utilisation d'une transaction SQL . 38
15 Fausse requête SQL récupérant les emails des clients, groupés par client et triés par *timestamp* le plus récent . . . 39
16 Requête SQL récupérant la liste des clients, avec pour chaque client la dernière version des attributs nom, prénom et email insérés . . . 39
17 Exemple de messages produits dans le *topic* des clients . . . 41
18 Boucle infinie de consommation des nouveaux événements dans `KafkaFlyManager` 42
19 Fonctions créant et mettant à jour les clients dans `KafkaFlyManager` . . . 43
20 Fonctions récupérant un client et son historique dans `KafkaFlyManager` . . . 43
21 Fonctions plus complexes, notamment sur les avions, de `KafkaFlyManager` . 44
22 Redéfinition de la fonction `query` dans `MySQLDebeziumFlyManager` . . . 46
23 Illustration de la fonction `seatNumber` de `MySQLDebeziumFlyManager` . . . 46
24 Fonction récupérant l'historique d'un client dans `MySQLDebeziumFlyManager` 47
25 Connexion à la CLI de ksqlDB . . . 48
26 Fichier de configuration initiale de ksqlDB . . . 48
27 Création du *stream* et de la table pour l'entité `Client` dans ksqlDB . . . 49
28 Exemple de requêtes ksqlDB *push* et avec fenêtrage temporel . . . 50
29 Conversion des *topics* Debezium dans `MySQLDebeziumKStreamsFlyManager` 51
30 Exemple de message produit dans le *stream* `client-stream` . . . 51
31 Fonction `client()` de `MySQLDebeziumKStreamsFlyManager` . . . 52

## Conventions typographiques

Lors de la rédaction de ce document, les conventions typographiques ci-dessous ont été adoptées.

- Tous les mots empruntés à la langue anglaise ou latine ont été écrits en *italique*.
- Toute référence à un nom de fichier (ou répertoire), un chemin d'accès, une utilisation de paramètre, variable, commande utilisable par l'utilisateur, ou extrait de code source est écrite avec une police d'écriture à `chasse fixe`.
- Tout extrait de fichier ou de code est écrit selon le format visible dans l'extrait 1 :

```
1 object Hello {
2     def main(args: Array[String]) = {
3         println("Hello, world")
4     }
5 }
```

Listing 1 – Hello world en Scala, exemple d'un extrait de code

## Structure du document

Le présent document commence par l'introduction, rappelant les motivations du projet et les buts à atteindre. Il aborde ensuite une analyse des différents *patterns*, architectures et technologies étudiées et / ou utilisées dans ce travail. La section suivante illustre la réalisation technique en elle-même, avec les différentes implémentations exposées. Finalement, le document se termine par une section de discussions et comparaisons des implémentations réalisées, de l'état du projet, des améliorations futures potentielles et une dernière section de conclusion. Les références se trouvent à la toute fin du document.

## Remerciements

Mes remerciements vont en premier lieu à ma compagne, Marie Bessat, pour son infinie patience, son soutien et ses encouragements. Je remercie aussi ma famille et mes proches pour leur soutien tout au long de mes études et lors de la réalisation de ce travail. Je tiens à remercier tous mes professeurs pour leurs enseignements et tout particulièrement M. Joël Cavat pour le suivi du projet et ses précieux conseils.

# Acronymes

**ACID** Atomicité, Cohérence, Isolation, Durabilité : les quatres propriétés d'une transaction dans les bases de données relationnelles, voir sous-section 2.1. 10, 51, 54

**API** *Application Programming Interface*, Interface de programmation : services offerts par un programme producteur à d'autres programmes consommateurs. 10, 16, 25, 28, 29, 45, 56

**CDC** *Change Data Capture*, Capture / monitoring des changements sur les données, voir sous-section 2.2. 11, 21

**CLI** *Command Line Interface*, Interface en ligne de commande : interface homme-machine en mode texte : l'utilisateur entre des commandes dans un terminal et l'ordinateur répond en exécutant les ordres de l'utilisateur et en affichant le résultat de l'opération. 25, 45, 46

**CQRS** *Command and Query Responsibility Segregation*, Patron de conception / architecture prônant la séparation d'une base de données en deux, une dédiée à l'écriture / mise à jour, l'autre uniquement pour la lecture, voir sous-section 2.4. 13, 23, 43, 48, 52, 56

**CRUD** *Create, Read, Update and Delete*, Les quatre opérations de manipulation qu'offre la plupart des bases de données : créer, lire, mettre à jour et supprimer. 8, 10, 11, 13, 14, 24, 29, 51

**CSV** *Comma-Separated Values*, Format de données texte disposant les données sous forme de tableau dont chaque colonne est séparée par un caractère de séparation, généralement une virgule. 38

**EA** Entité-Associations, Relatif au modèle entité-associations, c'est un modèle conceptuel décrivant la structure des données et les liens éventuels entre elles. 9, 27

**ETL** *Extract Transform Load*, processus d'extraction de données de différentes bases de données, de transformation (normalisation, filtres, etc.), et de chargement dans un entrepôt de données pour analyse ultérieure. 25

**FS** *File System*, Système de fichiers : organisation logique des fichiers sur le disque. 15

**GUI** *Graphical User Interface*, Interface graphique : moyen d'intéragir avec un logiciel où les contrôles et objets sont manipulables. S'oppose à l'interface en ligne de commande (CLI). 20

**HTTP** *Hypertext Transfer Protocol*, protocole de communication client-serveur quasi majoritaire pour le web, dans la couche réseau applicative. 25

**IATA** *International Air Transport Association*, Association internationale du transport aérien, en français. 27

**IoT** *Internet of Things*, expression "valise" désignant l'interconnexion des objets connectés par internet. 19, 20

**JDBC** *Java Database Connectivity*, API Java pour accéder à des bases de données, la plupart du temps relationnelles. 34

**JSON** *JavaScript Object Notation*, Format d'échange de données léger, facile à lire et écrire par les humains et les machines. 15, 21–23, 43, 46, 49

**JVM** *Java Virtual Machine*, Exécute le bytecode Java sur différents OS. 15

**OS** *Operating System*, Système d'exploitation : couche logicielle entre le matériel d'un ordinateur et les applications utilisateurs. Offre des abstractions pour la gestion des processus, des fichiers et des périphériques entre autres. 15

**REST** *Representational state transfer*, méthodologie de construction de services web, sans états, basée le plus souvent sur HTTP, avec une série de bonnes pratiques pour offrir une interopérabilité entre services. 25, 45

**SQL** *Structured Query Language*, Langage déclaratif pour l'interaction avec des bases de données relationnelles. 10, 23, 25, 35, 53

**TCP** *Transmission Control Protocol*, Protocole de transport sûr, établissant une connexion préalable entre les deux parties. 14

# 1 Introduction

## 1.1 Motivations

Avec l'augmentation de la puissance de calcul, de stockage et des capacités d'analyse des données quasi exponentielle qu'a connu les systèmes informatiques depuis des décennies cumulée à l'explosion des données produites par le web, le besoin et l'intérêt de conserver l'historique des changements survenus sur les données est devenu un argument stratégique. Plusieurs exemples de la vie quotidienne en témoignent. Un client d'une banque souhaite connaître l'évolution de son solde, savoir quand, combien et par qui l'argent a été déposé / retiré de son compte. Une administration aimerait retrouver la trace d'un document, savoir entre quelles mains il est passé et qui l'a édité. Un agriculteur voudrait déterminer la croissance de ses fruits en mesurant leur diamètre tout au long de la saison. Le modèle prédominant encore aujourd'hui pour les bases de données est celui relationnel, offrant cohérence et manipulation CRUD (*Create, Read, Update and Delete*) classique et efficace. Il peut se montrer pertinent même dans un contexte *big data*, mais il montre certaines limites lorsqu'il s'agit de monitorer des changements nombreux et massifs sur une même donnée. Dans ce contexte, nous allons explorer comment réaliser avec et sans base de données relationnelle un système gardant un historique des changements sur les données. Le rapport ainsi que le code source sont disponibles sur https://github.com/stevenliatti/mse-pa.

## 1.2 Buts

Les objectifs de ce projet d'approfondissement consistent à :

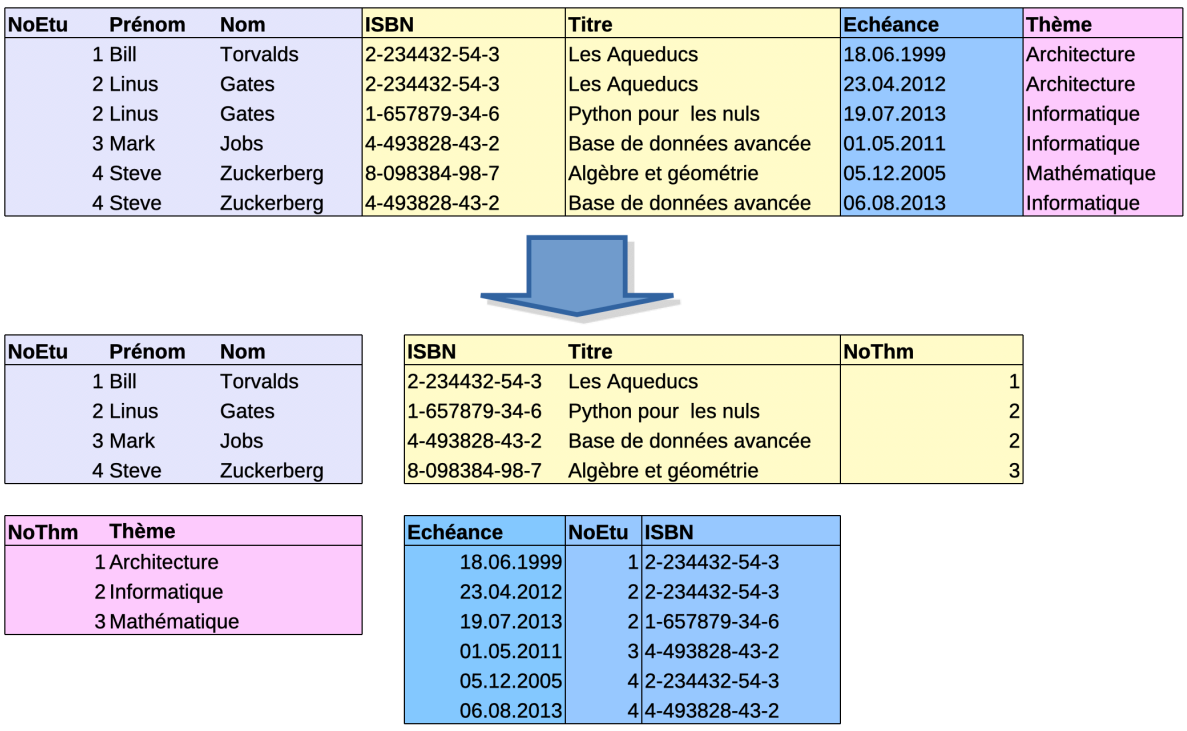
- Étudier les technologies, les moteurs de requêtes et les *patterns* existants mettant à disposition un *event store*.
- Proposer un *use case* et implémenter un contrat (interface / trait) selon les différents *patterns* et technologies.
- Montrer la transformation d'un modèle relationnel en un modèle orienté événements.
- Analyser la cohérence des données et les transactions qui permettent de garder une source de données en tant que source de confiance.
- Souligner les avantages et limitations des implémentations.
- Comparer les différentes implémentations entre elles et proposer les meilleurs cas d'utilisation.

# 2 Analyse architecturale et technologique

Dans cette section, nous allons présenter les technologies et concepts logiciels abordés et / ou utilisés dans ce travail. Les *patterns* présentés ici entrent dans la transition entre le monde relationnel et celui des événements.

## 2.1 Les bases de données relationnelles et MySQL

Les concepts et explications résumés de cette sous-section sont principalement tirés du cours de Systèmes de base de données de Joël Cavat [1]. Une base de données relationnelle est une source de données persistante. Elle est composée de tables (relations, au sens strict), elle-mêmes organisées en lignes et colonnes. Une table représente une entité ou une association du modèle EA (Entité-Association). Les bases de données évitent le plus possible les redondances d'informations. Mais il arrive que deux ou plusieurs tables partagent un attribut commun (associations). Plutôt que répéter l'information, des jointures peuvent être réalisées sur deux ou plusieurs tables pour relier l'information éparse dans plusieurs tables. La figure 1 illustre ce principe. Dans cet exemple, il est question d'une modélisation d'une bibliothèque, avec des emprunteurs (étudiants), des livres, des thèmes pour ces derniers et des dates de retour prévues. On voit la transformation d'une table, contenant toutes les informations pour chaque emprunt, avec redondance des informations, vers un modèle relationnel à plusieurs tables, séparant les préoccupations et supprimant ainsi la redondance.



| NoEtu | Prénom | Nom | ISBN | Titre | Echéance | Thème |
|---|---|---|---|---|---|---|
| 1 | Bill | Torvalds | 2-234432-54-3 | Les Aqueducs | 18.06.1999 | Architecture |
| 2 | Linus | Gates | 2-234432-54-3 | Les Aqueducs | 23.04.2012 | Architecture |
| 2 | Linus | Gates | 1-657879-34-6 | Python pour les nuls | 19.07.2013 | Informatique |
| 3 | Mark | Jobs | 4-493828-43-2 | Base de données avancée | 01.05.2011 | Informatique |
| 4 | Steve | Zuckerberg | 8-098384-98-7 | Algèbre et géométrie | 05.12.2005 | Mathématique |
| 4 | Steve | Zuckerberg | 4-493828-43-2 | Base de données avancée | 06.08.2013 | Informatique |

| NoEtu | Prénom | Nom |
|---|---|---|
| 1 | Bill | Torvalds |
| 2 | Linus | Gates |
| 3 | Mark | Jobs |
| 4 | Steve | Zuckerberg |

| ISBN | Titre | NoThm |
|---|---|---|
| 2-234432-54-3 | Les Aqueducs | 1 |
| 1-657879-34-6 | Python pour les nuls | 2 |
| 4-493828-43-2 | Base de données avancée | 2 |
| 8-098384-98-7 | Algèbre et géométrie | 3 |

| NoThm | Thème |
|---|---|
| 1 | Architecture |
| 2 | Informatique |
| 3 | Mathématique |

| Echéance | NoEtu | ISBN |
|---|---|---|
| 18.06.1999 | 1 | 2-234432-54-3 |
| 23.04.2012 | 2 | 2-234432-54-3 |
| 19.07.2013 | 2 | 1-657879-34-6 |
| 01.05.2011 | 3 | 4-493828-43-2 |
| 05.12.2005 | 4 | 2-234432-54-3 |
| 06.08.2013 | 4 | 4-493828-43-2 |

FIGURE 1 – Séparation de l'information entre plusieurs tables - Joël Cavat [1]

Une table a un nombre fixe de colonnes, donc d'attributs. Les attributs ne peuvent pas être représentés sous forme de listes. Les lignes des tables sont des enregistrements, différenciés les uns des autres par un identifiant unique, composé d'un ou plusieurs attributs, appelé clé primaire. Les clés primaires sont une des différentes contraintes d'intégrité offertes par les bases de données relationnelles. Même si on parle de contraintes, il s'agit là de contraintes "positives", on peut les voir comme garanties de cohérence des données. Deux autres mécanismes sont présents au service de la cohérence des données, les *triggers* et les transactions. Les *triggers* (ou "déclencheurs" en français) sont des vérifications effectuées lorsqu'un événement précis survient. Les transactions sont un mécanisme d'exclusion mutuelle garantissant qu'une séquence d'opérations sur des données soit réalisée de manière atomique, sans interruptions. Elles sont constituées de ces quatre propriétés, formant l'acronyme ACID :

- Atomicité : exécution sans interruptions, annulation de toutes les opérations si au moins une échoue
- Cohérence : les données restent dans un état cohérent avant et après la transaction
- Isolation : exclusion mutuelle oblige, une transaction appose un verrou sur les données qu'elle manipule
- Durabilité : le résultat d'une transaction est sauvée sur le disque

Les moteurs de bases de données relationnelles disposent de moyens d'intéragir avec les données, par API ou avec un langage de requêtes déclaratif, le *Structured Query Language* (SQL). Il offre entre autres les fonctionnalités CRUD minimales (pour rappel, les opérations créer, lire, mettre à jour et supprimer les données). La lecture des données est assez avancée et permet la sélection fine (`SELECT`), les filtres (`WHERE`), les jointures entre tables (`JOIN`), les regroupements (`GROUP BY`), le tri (`ORDER BY`) et le choix du nombre d'enregistrements retournés (`LIMIT`) entre autres. Il est possible de créer des index sur des attributs précis d'une table, pour accélérer la récupération en lecture. Dans ce travail, le moteur de bases de données relationnelles utilisé est MySQL [2]. Il est l'un des plus répandus, il est libre (GPL) et gratuit, performant et dispose des fonctionnalités basiques nécessaires. Il est concurrencé par d'autres moteurs très connus, comme MariaDB [3] (fork de MySQL), PostgreSQL [4] ou encore Microsoft SQL Server [5]. L'usage de MySQL aurait pu être remplacé par MariaDB ou PostgreSQL dans ce travail, tous trois offrant des fonctionnalités similaires et munis de licences libres, contrairement à Microsoft SQL Server.

Les bases de données relationnelles possèdent ainsi plusieurs avantages :

- Contraintes "positives" disponibles
- Transactions ACID
- Cohérence des données
- Source de données fiable, source de "vérité"

- Langage de requêtes performant et puissant

Mais également des inconvénients :

- Qui dit contraintes, dit moins de flexibilité : il est plus difficile d'adapter un schéma / modèle de données *a posteriori*
- Scalabilité verticale : les bases de données relationnelles sont difficilement distribuables, en cas de besoin de plus de puissance de calcul, on ne peut ajouter de machines / instances, mais plutôt augmenter la puissance de l'unique machine
- Conséquence du point précédent, la limite de la quantité de données gérable par une seule base de données relationnelle peut être rapidement atteint dans un contexte de "*big data*"
- Dans le cas du travail présent, la forte cohérence par la réduction de la redondance des données entre autres peut rendre plus complexe et moins intuitif l'élaboration d'un historique des changements ; les bases de données relationnelles demeurent plutôt dans l'univers CRUD "classique"

## 2.2 *Change Data Capture*

Pour obtenir un historique des versions des données, il faut effectuer le suivi des changements. C'est le principe de *Change Data Capture* (CDC) [6] ("capture des changements dans les données" en français). Dans les bases de données relationnelles, plusieurs stratégies sont possibles, la plupart consistant à ajouter des méta-données de changements aux enregistrements :

- Timestamps
- Numéro de versions
- Statuts
- Combinaison de timestamps, versions et statuts

De cette manière, on peut parcourir l'historique des changements en appliquant des filtres sur ces méta-données. Une autre manière de faire est d'ajouter une table par attribut changeant, contenant l'identifiant de la table principale, une indication supplémentaire unique (le timestamp par exemple) et la nouvelle version de la donnée. Dans les deux cas, ces deux méthodes ajoutent un *overhead* certain aux tables et / ou à la base de données. Nous verrons par la suite des techniques à l'aide de *message broker* (voir sous-section 2.5) qui résolvent ce problème.

## 2.3 *Event Sourcing*

Clément Héliou nous livre une excellente définition de l'*Event Sourcing* : "L'Event Sourcing est un pattern d'architecture qui propose de se concentrer sur la séquence de changements

d'état d'une application qui a amené cette dernière dans l'état où elle se trouve. L'idée n'est plus de savoir où nous sommes mais de garder trace du chemin parcouru pour y arriver." [7]. L'idée de base inhérente est relativement simple. Il est donc question de suite ou de séquence de changements survenus sur les données. Un exemple classique est celui du compte en banque, où le client peut voir la liste de ses transactions avec le montant (positif / négatif) et la date. La somme de ces événements depuis l'ouverture du compte retourne l'état courant du solde du compte. La table 1 illustre une séquence de transactions d'un compte bancaire.

| **date** | **crédit** | **débit** | **solde** |
|---|---|---|---|
| 2021-01-01 08 :10 :15 | 1000 | | 1000 |
| 2021-01-01 10 :56 :43 | | 200 | 800 |
| 2021-01-10 12 :47 :02 | | 100 | 700 |
| 2021-02-01 11 :30 :32 | 500 | | 1200 |
| 2021-02-04 08 :12 :23 | | 300 | 900 |
| 2021-02-17 22 :19 :09 | | 500 | 400 |
| 2021-03-01 08 :45 :53 | 700 | | 1100 |

TABLE 1 – Exemple d'une séquence de transactions d'un compte bancaire

C'est exactement le comportement recherché lorsque l'historique des changements des données est voulu : à travers ces événements, la "vie" de la donnée peut être observée, de sa création à sa destruction éventuelle, en passant par un nombre indéfini d'états intermédiaires. Se reposer sur une séquence d'événements plutôt que sur le dernier état courant a plusieurs avantages :

- Un événement est facilement compréhensible par l'humain, on peut le traduire en langage naturel ("Telle entité a réalisé telle action sur telle information")
- Insertions concurrentes possibles, l'ordre reçu des événements faisant foi
- Un événement est immutable, il est ajouté en fin de liste des événements, cet aspect favorise l'asynchronisme (et l'augmentation de performances qui en suit)
- Les bugs et autres mauvaises manipulations sont plus facilement détectables, on peut rejouer l'historique des événements
- Découplage fort des services abonnés au même *event store*

Néanmoins, il est sujet à quelques défauts :

- En cas de changement dans le format des événements, des difficultés peuvent apparaître : il est important d'utiliser un format d'événements simple et sans surplus d'informations
- La cohérence des données, en lecture, n'est pas forcément garantie
- L'*Event Sourcing* n'est pas une base de données en lui-même, il faut construire un système par-dessus pour exécuter les requêtes

Ces articles de Microsoft [8], Martin Fowler [9] et Chris Richardson [10] approfondissent le sujet et ont servi de base pour ces explications.

## 2.4 CQRS *pattern*

L'acronyme CQRS signifie *Command and Query Responsibility Segregation*. Comme son nom l'indique, ce patron de conception (*pattern*) met en avant la notion la séparation des responsabilités entre les "commandes" et les "requêtes", constituants les opérations CRUD sur les données. "Commandes" et "requêtes" font ici, respectivement, référence aux côtés *create*, *update* et *delete* d'une part et de *read* d'autre part. Plutôt qu'utiliser une seule base de données, ou plus généralement une seule source de données, pour les quatre opérations, une nette séparation entre les accès en écriture et en lecture est mise en place, avec la possibilité d'utiliser deux modèles différents (sans obligation toutefois). La figure 2 illustre le principe de ce *pattern*. Les parties en traitillés indiquent des éléments potentiellement optionnels, il n'y a pas d'obligation stricte d'utiliser un *message broker*, mais on y voit clairement une volonté de séparer les accès en écriture des accès en lecture aux données.

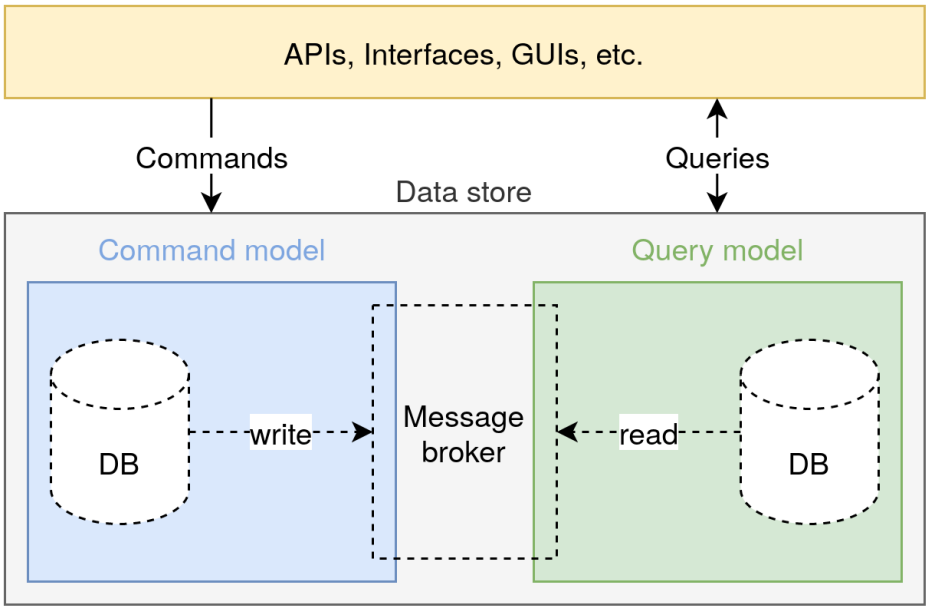



FIGURE 2 – Schéma de principe du *pattern* CQRS

Par rapport à un CRUD "classique", CQRS offre plusieurs avantages :

- Scalabilité séparée : côtés lecture et écriture peuvent être mis à l'échelle de manière indépendante, selon les besoins
- Modèles logiques différents : un adapté pour l'écriture, l'autre pour la lecture
- Requêtes en lecture simplifiées : en créant des vues matérialisées, on évite l'aggrégation de données (`joins` relationnels)

- Non concurrence entre les opérations de lecture et d'écriture
- Performances accrues et isolation, grâce à l'optimisation des modèles et à la répartition des charges pour chaque côté
- Gestion des droits et accès clair facilités : les utilisateurs et applications accédant aux données ont des droits précis, réduction des risques d'accès non permis
- Facilité à rendre l'écriture asynchrone : commandes placées dans une file
- Partie lecture ne modifie pas les données, pas de risque de mauvaise manipulation

Il faut néanmoins avoir ces défauts en tête :

- Augmentation de la complexité : il n'y a plus une, mais au moins deux sources de données à gérer et manipuler
- Cohérence des données : en séparant fortement la lecture de l'écriture, il y a un risque que les requêtes effectuées retournent des données périmées, car le côté lecture n'aurait pas eu le temps de se mettre à jour (selon les cas, ce défaut est critique ou non)

Le *pattern* CQRS n'implique pas forcément l'obtention d'un historique d'événements. On peut très bien l'appliquer sur un modèle CRUD classique. Mais en l'associant au *pattern Event Sourcing* (voir sous-section 2.3), un *message broker* s'insère parfaitement comme partie écriture : hautes performances en écriture, asynchronisme, écritures parallèles, etc. Ce modèle est en partie implémenté dans ce travail (voir section 3). Les explications ci-dessus sont tirées des articles de Microsoft [11], de Chris Richardson [12], d'Octo [13] [14] et de Martin Fowler [15]. Ces différents articles approfondissent le concept.

## 2.5 Le *message broker* Kafka

Basiquement, un *message broker* peut être défini comme un système qui écoute sur des messages entrants, provenant de programmes "sources" (*source*), appelés "producteurs", et qui les relaie à d'autres programmes "puits" (*sink*) dits également "consommateurs". Les termes "source" et "puit" parlent d'eux-mêmes, on arrive facilement à s'imaginer la direction que suit l'information (de la "source" vers le "puit"). L'unité de base utilisée au sein d'un *message broker* est donc un "message", appelé aussi "événement" ou "enregistrement" (*event* ou *record*). Kafka [16] est un *message broker* (ou agent de messages en français) performant, fiable et hautement scalable (distribué). Il est écrit en Java et Scala et est sous licence Apache. Il offre, entre autres, la possibilité de "publier" un flux (*stream*) de messages et / ou de s'y "abonner", de stocker les messages reçus sur disque dur (avec une durée d'expiration optionnelle) et de traiter puis transformer un flux de messages en un autre, le tout avec une latence faible (le protocole d'échange binaire de Kafka est construit par dessus TCP). Il dispose de tout un écosystème construit sur ses bases (Kafka Connect, Kafka Streams, ksqlDB). Les sous-sections suivantes vont exposer quelques concepts clé de Kafka et des *message broker* en général. Ces

concepts sont tirés de la documentation officielle de Kafka, en particulier de la partie Design [17].

### 2.5.1 Messages et persistance

Un message Kafka (ou *record*) indique qu'un nouvel événement a eu lieu, que quelque chose "s'est passé". Il est constitué d'un en-tête (*header*) optionnel, d'une clé (*key*), d'une valeur (*value*), d'un *timestamp* et d'un *offset* (correspondant à la position du message dans la partition, voir sous-section 2.5.5). Les *header*, *key* et *value* sont des tableaux d'octets de taille variable, laissant ainsi la liberté d'y stocker ce qu'on veut. Le message est finalement une structure de donnée assez bas niveau, simple, mais sans contrainte sur son usage. Le listing 2 montre un exemple d'un message qui pourrait être échangé dans Kafka, en JSON comme format arbitraire.

```
{
  "key": "Fred",
  "value": {
    "info": "Is online",
    "ip": "123.123.123.123"
  },
  "timestamp": 1616863271,
  "offset": 42
}
```

Listing 2 – Exemple de message Kafka, au format JSON

Les messages sont stockés sur disque dur. On pourrait penser qu'un système vantant une latence faible et des hautes performances ferait un usage intensif de la mémoire vive de l'OS plutôt qu'utiliser le disque dur, qui, même dans les meilleurs cas, demeure bien plus lent d'accès que la mémoire vive. Mais il ne faut pas oublier que Kafka est exécuté à travers la JVM, le *garbage collector* peut s'avérer imprévisible, surtout lorsqu'une grande quantité de mémoire est allouée à la JVM. Le parti pris de Kafka est simple : écrire le message directement sur le disque, réduisant fortement le risque de le perdre et se reposer sur le cache de l'*operating system* OS et / ou du *file system* FS. Une explication plus fournie se trouve dans la documentation de Kafka, au paragraphe Persistence [18].

### 2.5.2 Topics

Un *topic* (qu'on pourrait traduire par "sujet" en français) est un regroupement de messages ou événements partageant une même thématique. On peut faire l'analogie avec le FS : un *topic* est un répertoire et les messages sont les fichiers qu'il contient. Il peut y avoir zéro, un ou

N producteurs et / ou consommateurs reliés à un *topic*. Un *topic* peut être distribué sur plusieurs partitions (voir sous-section 2.5.5) et être répliqué, dans un souci de redondance et protection des données. La durée de rétention des messages est configurable par *topic* et peut être définie selon une fourchette temporelle (par exemple, un jour, une semaine, etc.), par défaut "infinie". Contrairement à d'autres *message broker* (par exemple RabbitMQ), Kafka autorise les consommateurs à relire l'historique d'un *topic*, à partir du début ou d'un moment particulier (*offset*).

### 2.5.3 Producteurs

Un programme producteur est un programme qui émet des messages, dans un ou plusieurs *topics*. C'est la partie la plus simple de l'écosystème Kafka, un producteur génère un message (logique), le sérialise dans le format voulu, se connecte au serveur (*broker*) et envoie le message. Le producteur peut être configuré pour envoyer les messages par lot (*batch*), pour réduire le nombre de requêtes. Ce traitement par lot peut être configuré soit au nombre de messages (par exemple, si 10 messages sont accumulés, la requête peut être effectuée), soit par le temps d'attente avant d'envoyer la requête. Kafka offre de nombreuses APIs pour différents langages de programmation pour produire et consommer des messages (Kafka Connect). Celle officielle demeure néanmoins celle pour Java. Au listing 3, on peut apercevoir un exemple simple d'un code producteur en Scala, provenant du tutoriel de Loïc Divad [19]. Un `randomRecord` de type `ProducerRecord[Key, Rating]` est initialisé, avec le *topic* de destination, la clé, la valeur (*rating*) et le *timestamp* et l'envoie au *broker* (connexion réalisée au préalable).

```
1  val rating: Short = Random.nextInt(5).toShort
2  val eventTime: Long = Instant.now.toEpochMilli
3  val (showKey, showValue) = // ....
4  val genHeader =  new RecordHeader("generator-id", s"..." getBytes)
5  val showHeader = new RecordHeader("show-details", s"..." getBytes)
6
7  val randomRecord = new ProducerRecord[Key, Rating](
8    "topic-name2",
9    null, // let the defaultPartitioner do its job
10   eventTime,
11   showKey,
12   Rating(randomUUID().toString.take(8), rating),
13   new RecordHeaders(Iterable[Header](genHeader, showHeader).asJava)
14 )
15
16 producer2 send randomRecord
```

Listing 3 – Exemple de producteur Kafka en Scala - Loïc Divad [19]

### 2.5.4 Consommateurs

Un programme consommateur est le pendant du producteur. Il se connecte à un ou plusieurs *topics* et attend de recevoir des messages. Le consommateur est l'instigateur de la consommation d'un ou plusieurs messages : c'est lui qui demande au *broker* (*pull*) le message voulu (avec l'*offset*) et reçoit en retour les nouveaux messages à partir de cet *offset*, et non le *broker* qui envoit au consommateur (*push*) les derniers messages en date. De cette manière, on réduit la charge sur le *broker* et le consommateur peut consommer les messages "à son rythme", il ne risque pas d'être débordé. Un consommateur peut s'abonner à un *topic* à partir du dernier message en date ou depuis le début de l'historique des événements. Le listing 4 illustre un simple exemple d'un consommateur en Scala, initialisant un `KafkaConsumer[String, String]`, s'abonnant à un seul *topic* `mytopic`, consommant les messages depuis le début et, de manière "infinie", attendant sur un nouveau message (vérification toutes les 100 millisecondes) pour l'afficher sur la sortie standard.

```scala
object ConsumerExample extends App {
  val topic = "mytopic"
  val properties = new Properties()
  properties.put(.....)

  val consumer = new KafkaConsumer[String, String](properties)
  consumer.subscribe(Collections.singletonList(topic))

  consumer.poll(0) // 1er poll bidon pour rejoindre le topic
  consumer.seekToBeginning(consumer.assignment)

  while (true) {
    val records = consumer.poll(100)
    for (record <- records.asScala) {
      println(record)
    }
  }
}
```

Listing 4 – Exemple de consommateur Kafka en Scala

### 2.5.5 Partitions

Kafka étant scalable de manière horizontale (*id est* ajout de machines *versus* augmentation de la puissance d'une unique machine), plusieurs serveurs *broker* peuvent co-exister. Cela implique qu'un *topic* peut se retrouver segmenté, on dit qu'il est partitionné. Ce partitionnement autorise ainsi la lecture et écriture simultanée d'un même *topic*, mais sur des partitions différentes, au bénéfice de performances en hausse. Pour garantir que la consommation des messages se déroule dans l'ordre dans lequel il a été inséré dans la *partition*, Kafka place les messages avec la même clé dans la même partition. La figure 3 donne un exemple de ce comportement. Ce *topic* contient quatre partitions. Deux producteurs écrivent des messages simultanément dans différentes partitions. Les messages ayant la même clé (ici représentés par les couleurs) atterissent dans la même partition.

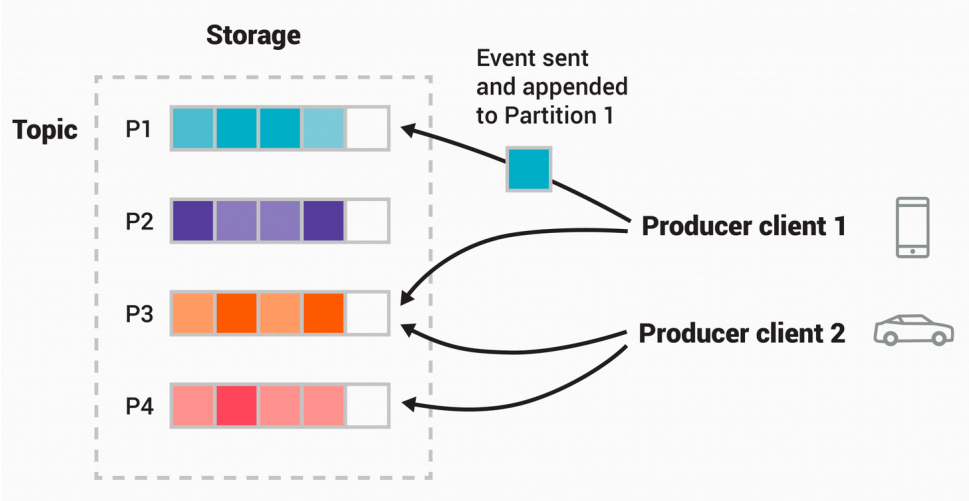



FIGURE 3 – Illustration de la répartition des messages dans différentes partitions - Kafka documentation [20]

### 2.5.6 Streams

Les *streams* Kafka sont similaires aux *streams* dont dispose Scala ou Java, pour rappel des séquences de données infinies et paresseuses (*lazy*), mais les données d'entrée et les résultats proviennent tous deux de *topics* Kafka. Un *stream* Kafka contient un ou plusieurs consommateurs et producteurs : il s'abonne à un *topic*, lis tous les messages, y applique les filtres et autres opérations désirés (fonctions similaires à `map`, `filter`, `groupBy`, etc. en Scala) et les publie dans un autre *topic*. Ainsi, on dispose de primitives semblables à celles de Java ou Scala mais avec les bénéfices du stockage et de l'efficience d'instances Kafka distribuées. Le listing 5 montre un exemple simple d'un *stream* Kafka, provenant de la documentation Kafka sur les *streams* [21]. Il reçoit en entrée des lignes de texte (par exemple une ligne par message) provenant du *topic* "TextLinesTopic". Pour chacune de ces lignes, il les passe en lettres minuscules et les convertit en une liste de mots (séparation de la ligne en mots).

Finalement, il les groupe par mot et compte les occurences de ce mot. Le résultat final est publié dans le *topic* "WordsWithCountsTopic".

```
object WordCountApplication extends App {
  val props: Properties = ....

  val builder: StreamsBuilder = new StreamsBuilder
  val textLines: KStream[String, String] = builder.stream[String,
  String]("TextLinesTopic")
  val wordCounts: KTable[String, Long] = textLines
    .flatMapValues(textLine => textLine.toLowerCase.split("\\W+"))
    .groupBy((_, word) => word)
    .count()(Materialized.as("counts-store"))
  wordCounts.toStream.to("WordsWithCountsTopic")

  val streams: KafkaStreams = new KafkaStreams(builder.build(), props)
  streams.start()
}
```

Listing 5 – Exemple de *stream* Kafka en Scala - Kafka Streams Documentation [21]

Étant donné que ksqlDB repose sur les Kafka Streams, les concepts plus avancés de ces deux entités sont décrits dans la section dédiée à ksqlDB (voir sous-section 2.8), pour éviter les doublons.

## 2.6 Alternatives à Kafka

Les géants du web proposent tous leur alternative à Kafka : Amazon Kinesis [22], Google Pub/Sub [23] ou encore Azure Service Bus [24]. Ces solutions, bien que potentiellement valables et performantes, ont été écartées d'office. Elles ont le défaut inhérent d'enfermer l'utilisateur dans leur écosystème, nous favorisons les solutions un peu plus libres et indépendantes. Il existe tout de même des alternatives ouvertes à Kafka que nous allons passer en revue rapidement.

### 2.6.1 ActiveMQ

ActiveMQ [25] est un *message broker* écrit en Java. Il supporte plusieurs protocoles de sérialisation / désérialisation de messages tels que AMQP (*Advanced Message Queuing Protocol*), STOMP (*Simple Text Oriented Messaging Protocol*), MQTT (*Message Queuing Telemetry Transport*, orienté IoT). ActiveMQ est un projet plus ancien que Kafka. Il est plus orienté queue

de priorité que producteurs-consommateurs. Par défaut ActiveMQ ne garde pas les messages une fois consommés, activer cette fonctionnalité fait qu'il devient moins scalable que Kafka. Il offre plus de possibilités pour ce qui concerne le *routing* des messages entre producteurs et consommateurs. Il ne garantit pas le respect de l'ordre des messages. De manière générale, ActiveMQ est un *broker* plus "intelligent", il peut acheminer les messages de manière plus complexe, supporte plusieurs protocoles et repose sur le principe du *push* des messages vers les instances consommatrices plutôt sur le principe du *pull*, ce qui fait de lui un bon candidat en tant que pilier d'un système IoT par exemple. L'article de blog de Victor Alekseev [26] offre une comparaison détaillée entre ActiveMQ et Kafka.

### 2.6.2 RabbitMQ

RabbitMQ [27] est un *message broker* écrit en Erlang. Comme ActiveMQ, il supporte plusieurs protocoles d'échange de messages (AMQP, MQTT, STOMP, etc.). Grâce à la conception intrinsèque d'Erlang, il est hautement scalable et parfaitement adapté pour gérer de grosses charges. Il offre également plusieurs possibilités de *routing* des messages. Comparé à ActiveMQ, il serait un bon candidat grâce à ses performances. Néanmoins, il n'offre pas la possibilité de rejouer l'historique. Une fois qu'un message est consommé (avec possibilité d'envoyer un *acknowledge*), il est définitivement retiré de la queue. On voit là son orientation claire vers l'IoT ou les queues de priorités, surtout avec sa fonctionnalité de confirmation de réception d'un message. Lovisa johansson livre une comparaison détaillée dans son article de blog [28].

### 2.6.3 Logstash et suite ELK

La suite ELK (Elasticsearch, Logstash et Kibana) [29] est un moteur de recherche et d'analyse distribué. Le moteur en lui-même est Elasticsearch, doté de capacités d'indexations importantes (index, kd-tree, etc.). Kibana est une GUI pour visualiser les informations sous forme de graphiques et agis comme interface pour exécuter les requêtes vers Elasticsearch ou monitorer le système entier. Logstash [30] est le point d'entrée du système. Il permet de lire les données provenant de multiples sources et de les unifier pour les présenter à Elasticsearch. Il fait donc une transformation entre formats prédéfinis (par exemple : entre les logs MySQL vers des emails). De ce fait, Logstash n'est pas un *message broker* à proprement parler. Il pourrait s'interfacer entre une application et un *message broker* par exemple, pour convertir les données vers le format désiré. C'est d'ailleurs ce que proposent entre autres Suyog Rao et Tal Levy dans leur article [31].

### 2.6.4 Choix final porté sur Kafka

ActiveMQ a été écarté pour son manque de documentation (pas jusqu'à dire que le projet est mort, mais il est beaucoup moins populaire) et ses comportements par défaut de non-

sauvegarde des messages et sa scalabilité moindre. RabbitMQ a aussi été écarté à cause de la non-rétention définitive de messages (pour rappel, le *broker* peut se substituer à une base de données dans notre *use case*). ELK et Logstash sont intéressants, mais sont plutôt pensés pour la recherche et indexation, un cas différent du notre. Même si les solutions concurrentes peuvent être pertinentes dans certains cas (voir cet article de Kai Waehner [32]), il est dommage de ne pas utiliser Kafka dans un contexte d'*event sourcing*. Kafka est puissant pour les principes suivants : scalabilité, efficience, sauvegarde sur disque, *streams*, *broker* "simple" mais efficace. Il se concentre sur la performance et la résilience, sans complexifier ses fonctionnalités de base. Il offre en plus un écosystème et des intégrations tout-à-fait pertinentes dans le cadre ce travail (ksqlDB et Debezium). Kafka est donc un socle solide qui va nous accompagner dans certaines implémentations et expériences.

## 2.7 Debezium

Debezium [33] est une plateforme de CDC. Debezium permet de se connecter à diverses bases de données (MySQL, PostgreSQL, MongoDB) et capture les changements survenus sur les données. Ces changements sont ensuite publiés dans Kafka, Debezium ayant son propre connecteur Kafka (Kafka Connect). La figure 4 illustre l'architecture et une proposition de pipeline d'utilisation pour Debezium.

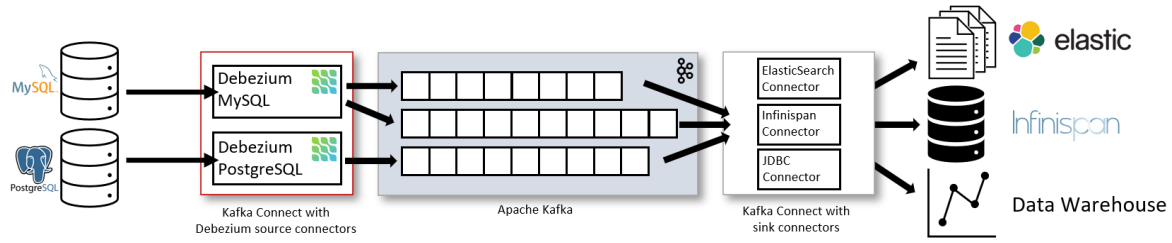



FIGURE 4 – Architecture et proposition de pipeline de Debezium - Debezium [34]

Dans le cas de MySQL (base de données qui nous intéresse dans ce travail), les comportements exacts sont les suivants : Debezium surveille le `binlog` de MySQL [35], le journal des requêtes et commandes de MySQL. Chaque commande comportant un `INSERT`, `UPDATE` ou `DELETE` est capturée et écrite dans Kafka. Il y a un *matching* un-à-un entre une table MySQL et un *topic* Kafka, de sorte que tous les changements d'un survenus dans une table se retrouve dans le *topic* correspondant. Le format de production Kafka utilisé par défaut par Debezium est le JSON, mais il est également possible d'utiliser Avro [36], un format binaire. Un message Debezium consiste en deux parties, la clé, visible sur le listing 6 et la valeur, visible sur le listing 7, ces deux parties correspondant aux clés et valeurs d'un *topic* Kafka. La clé a elle-même deux parties, le `schema` et le `payload`. Le `schema` décrit la structure du `payload`, dans ce cas une `struct` contenant un `dbserver.fly.Client.Key`. Le `payload` contient le ou les

champs composant la clé primaire de la table MySQL correspondante, dans notre exemple un `id` ayant comme valeur `1000`.

```
{
  "schema": {
    "type": "struct",
    "fields": [
      {
        "type": "int32",
        "optional": false,
        "field": "id"
      }
    ],
    "optional": false,
    "name": "dbserver.fly.Client.Key"
  },
  "payload": {
    "id": 1000
  }
}
```

Listing 6 – Exemple d'une clé d'un événement publié dans Kafka depuis Debezium, au format JSON

La partie concernant la valeur est elle aussi divisée en deux parties, `schema` et `payload`. Le `schema` contient de nombreuses informations redondantes sur la structure, dans le cas ou le format des données évoluerait. Le `payload`, lui, contient quatre champs importants :

- `op` : ce champ indique l'opération effectuée lors de cet événement, ici valant `c` (pour "*create*"), car notre exemple traite un `INSERT` ; les deux autres possibilités sont `u` (`UPDATE`) ou `d` (`DELETE`)
- `before` : ce champ indique la valeur des champs de l'enregistrement MySQL avant l'opération ; dans ce cas de création, la valeur précédente était nulle, alors que dans les cas de mise à jour ou de suppression, il y aurait l'ancienne valeur
- `after` : ce champ indique la valeur des champs de l'enregistrement MySQL après l'opération ; dans ce cas de création (et de mise à jour), la valeur suivante est la nouvelle valeur, ou valeur actuelle, alors que dans le cas de suppression, elle serait nulle
- `ts_ms` : ce champ indique le *timestamp* UNIX en millisecondes de l'opération

En lisant les valeurs des champs `op`, `after` et `before`, il est assez simple de construire un historique de l'évolution de l'enregistrement au cours du temps.

```
{
  "schema": {
    "type": "struct",
    "fields": [...],
    "optional": false,
    "name": "dbserver.fly.Client.Envelope"
  },
  "payload": {
    "before": null,
    "after": {
      "id": 1000,
      "firstname": "Klement",
      "lastname": "Scothorne",
      "email": "kscothornerr@squarespace.com"
    },
    "source": {...},
    "op": "c",
    "ts_ms": 1620835101836,
    "transaction": null
  }
}
```

Listing 7 – Exemple d'une valeur d'un événement publié dans Kafka depuis Debezium, au format JSON

Debezium semble donc la solution parfaite pour implémenter les *patterns Event Sourcing* et CQRS.

## 2.8 ksqlDB

ksqlDB [37] est une base de données optimisée pour les applications de *stream processing*. Elle est entièrement construite par-dessus Kafka Streams (2.5.6), la persistance des données étant faite dans des *topics* Kafka standards. ksqlDB repose donc évidemment sur les événements, qui constituent son unité de base. ksqlDB permet d'exécuter des commandes et requêtes à l'aide d'un pseudo SQL sur deux structures complémentaires : les *streams* et les tables. Un *stream* ksqlDB est l'équivalent d'un *topic* Kafka. Il contient donc une série d'événements (ou messages), munis d'une clé implicite ou explicite. Un ou plusieurs événements avec la même clé peuvent coexister dans le *stream*. Les tables quant à elles s'approchent de leurs homonymes des bases de données relationnelles, avec quelques nuances. Une table est

construite par-dessus un ou plusieurs *streams*, mais là où elle se différencie de ces derniers, c'est qu'elle ne garde en mémoire que le dernier état d'une information, là où le *stream* retient la séquence des changements. Si plusieurs enregistrements partagent une même clé, la table ne retient que le plus récent. Grâce aux tables, des vues matérialisées peuvent être définies, des projections des données basées uniquement sur la différence entre un événement et son prédécesseur plutôt que sur l'entièreté des enregistrements. C'est ce qui est décrit comme la "dualité des *streams* et des tables" [38] [39], il est possible de dériver une table à partir d'un *stream* et inversement. La table 2 illustre cette dualité (reprise de Michael Noll [38]).

| | **Stream** | **Table** |
|---|---|---|
| First event with key `bob` arrives | Insert | Insert |
| Another event with key `bob` arrives | Insert | Update |
| Event with key `bob` and value `null` arrives | Insert | Delete |
| Event with key `null` arrives | Insert | <ignored> |

TABLE 2 – Comparaison des effets de chaque opération CRUD sur les *streams* et les tables - Michael Noll [38]

Les rôles des requêtes et données sont inversés entre les bases de données relationnelles et ksqlDB : dans une base de données classique, les requêtes sont dites "actives" et les données "passives", car les premières sont exécutées à la volée sur l'entièreté des secondes. Dans ksqlDB, c'est l'inverse, les requêtes sont dites "passives" et les données "actives", ces dernières provenant d'un flux asynchrone changeant continuellement. La figure 5 décrit cette idée.

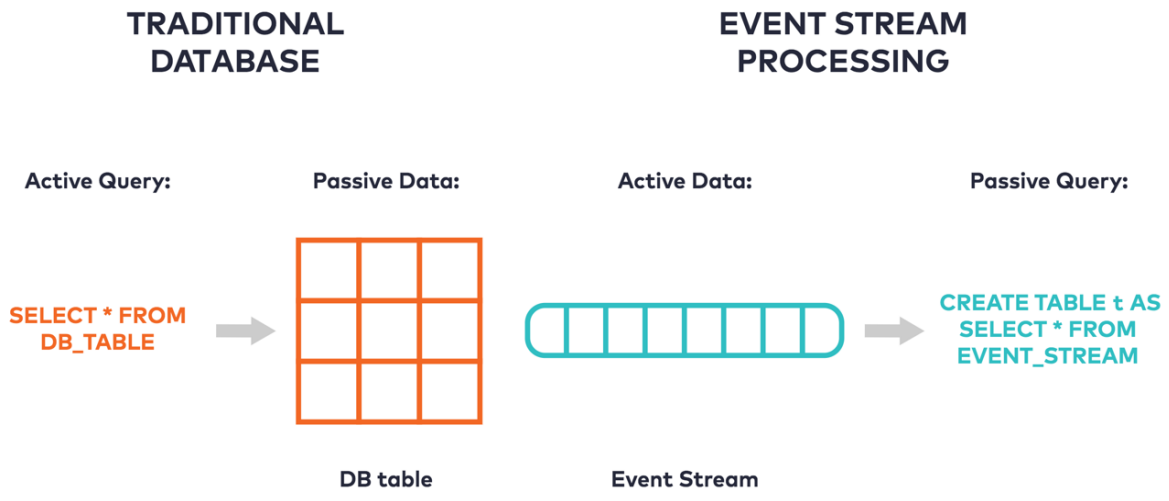



FIGURE 5 – Confrontation entre bases de données relationnelles et ksqlDB sur les requêtes et données - Confluent Inc. [40]

Voyons maintenant un peu plus en détails le langage de requêtes. On peut donc créer / supprimer des tables et des *streams* (`CREATE` et `DROP`) et insérer des données à l'intérieur

(`INSERT`). En revanche, comme rétention des données se fait dans des *topics* Kafka, les enregistrements sont immutables, il n'y a pas d'opérations de mise à jour (`UPDATE`) ou de suppression (`DELETE`) disponibles. Concernant les opérations de sélection (`SELECT`), on peut réaliser pratiquement toutes les opérations possibles avec MySQL et consorts. Il est possible de réaliser des jointures *streams-streams*, *streams*-tables ou tables-tables, mais uniquement sur les attributs constituant la clé primaire de l'enregistrement. Deux types de requêtes sont par contre "innovants" par rapport au SQL classique, ce sont les *push queries* et *window queries*. Les *push queries*, comme leur nom l'indique, sont envoyées par le serveur ksqlDB (au client qui les demandent) lorsqu'un nouvel événement survient et qu'il correspond aux critères de la requête. Le listing 8 illustre un exemple d'une telle requête, dans ce cas on demande de recevoir les clients dont le nom de famille commence par un "A" à l'instant où ils sont ajoutés au *stream* `clients`.

```
1 SELECT * FROM clients
2 WHERE SUBSTRING(firstname, 1, 1) = 'A'
3 EMIT CHANGES;
```

Listing 8 – Exemple de requête *push* ksqlDB

Les *window queries* sont des requêtes qui peuvent être déclenchées en fonction de variables temporelles, à l'aide des mots-clé `WINDOWSTART` et `WINDOWEND`. On crée donc une "fenêtre" dans laquelle tout enregistrement correspondant aux critères de la requête et aux contraintes temporelles sera retourné. Grâce aux *push queries* et aux *window queries*, ksqlDB permet de monter une application qui réagit aux données en temps réel assez aisément. ksqlDB dispose d'une CLI, principalement dans un but de test et configuation, mais un client Java et une API HTTP REST sont également disponibles.

Pour conclure cette présentation de ksqlDB, les figures 6 et 7 illustrent la transition et les bénéfices que peut offrir ksqlDB. Sur la première image on distingue ce qu'on pourrait appeler un pipeline "classique" de traitement de données ETL, avec plusieurs applications écrivant dans Kafka (*Extract*), un ou plusieurs *frameworks* d'analyse des données "*offline*", comme Apache Spark [41], transformant les données (*Transform*) depuis et vers Kafka et finalement réinsérant les données dans d'autres bases de données ou *frameworks* (*Load*) pensés pour la visualisation et interaction avec des applications utilisateurs. La seconde image montre que ksqlDB peut se charger du processus ETL de bout en bout en se substituant à ces différents *frameworks* grâce à ses multiples connecteurs, son *stream processing* et ses vues matérialisées. Les tutoriels sur l'élaboration d'une vue en tant que cache [42], sur la présentation d'un pipeline ETL [43] et sur le développement d'un microservice d'envoi d'emails [44] présents sur la documentation de ksqlDB ont été d'une aide précieuse pour la compréhension et prise en main de ksqlDB.

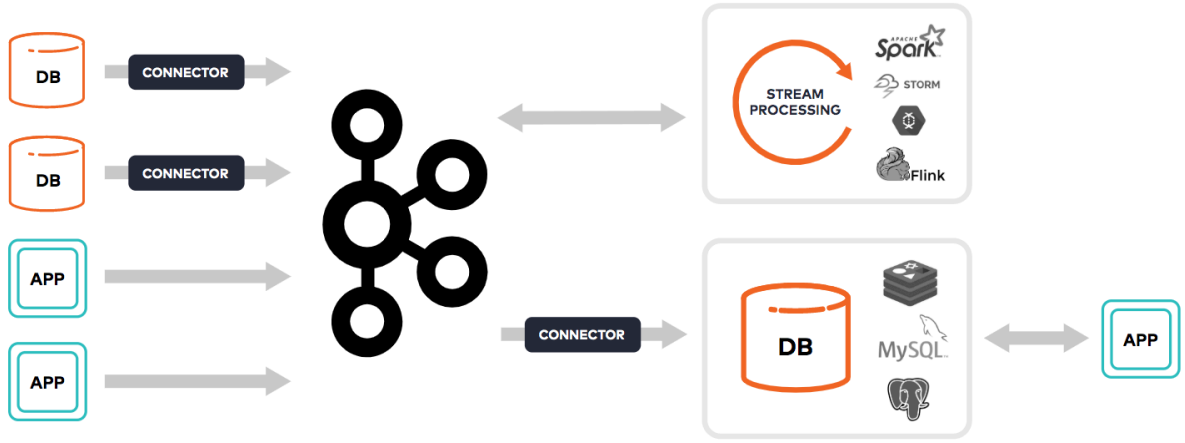



FIGURE 6 – Pipeline de traitement de données *ante* ksqlDB - Confluent Inc. [40]


FIGURE 7 – Pipeline de traitement de données avec ksqlDB - Confluent Inc. [40]

# 3 Implémentations

## 3.1 Modèle initial, génération de données et base commune

### 3.1.1 Modèle initial

Le modèle qui a servi de base pour les diverses implémentations a comme source d'inspiration un modèle présenté par Joël Cavat dans son cours de Systèmes de base de données [1], soit une compagnie aérienne factice, gérant des vols (`Flight`) avec une heure de départ, un aéroport de départ et d'arrivée. Ces aéroports (`Airport`) ont un nom et sont identifiés par un code IATA (AITA en français) [45] à trois lettres. Les vols sont effectués par des avions (`Aircraft`), d'une marque donnée et disposant d'un nombre de sièges fixe. Les vols accueillent évidemment des clients (`Client`), avec comme attributs un prénom, un nom et une adresse email. L'association entre les clients et les vols est une réservation (`Booking`), avec comme attribut le numéro du siège. Le modèle est à la fois simple, étant constitué de peu de types d'entités, mais également suffisamment complexe pour avoir des contraintes d'intégrité (un siège ne peut être attribué qu'à un seul client sur un vol par exemple) et des dépendances fonctionnelles (connaissant un numéro de vol et un numéro de siège on peut retrouver un client par exemple) qui le rendent "intéressant" comme sujet d'étude. La figure 8 illustre le diagramme entité-associations (EA) de ce modèle. On y distingue notamment l'association *booking* entre `Client` et `Flight` ainsi que la cardinalité associée, plusieurs-à-plusieurs.

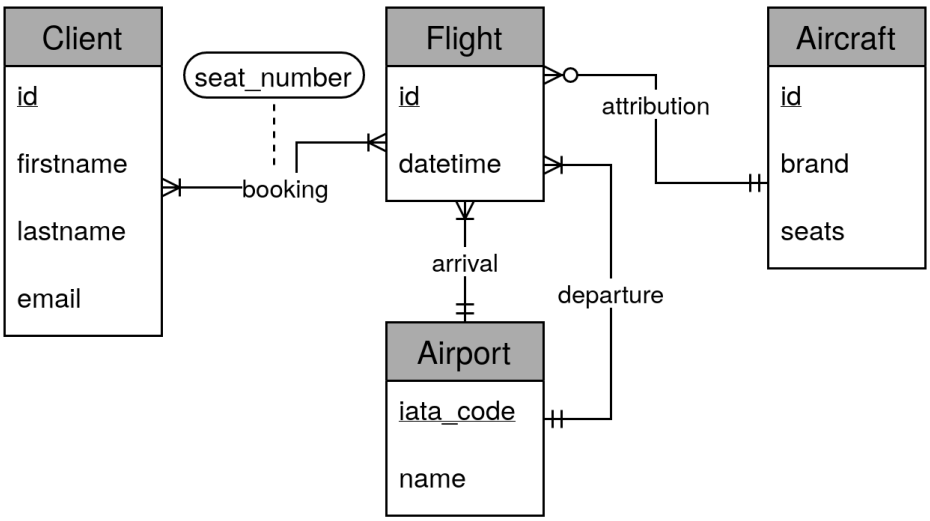



FIGURE 8 – Diagramme entité-associations initial

C'est précisément cette association qui nous intéresse : dans le monde relationnel, ces associations (car il y en a plusieurs) sont modélisées dans une table intermédiaire, comme on peut le voir sur la figure 9, montrant la transformation du modèle conceptuel (EA) au modèle logique. Dans notre cas pratique, le système doit pouvoir gérer le changement de siège

d'un client en gardant l'historique de ces changements. Il n'est pas évident, ou du moins pas pratique, de modéliser l'historique des changements d'un tel enregistrement au sein d'une base de données relationnelle.

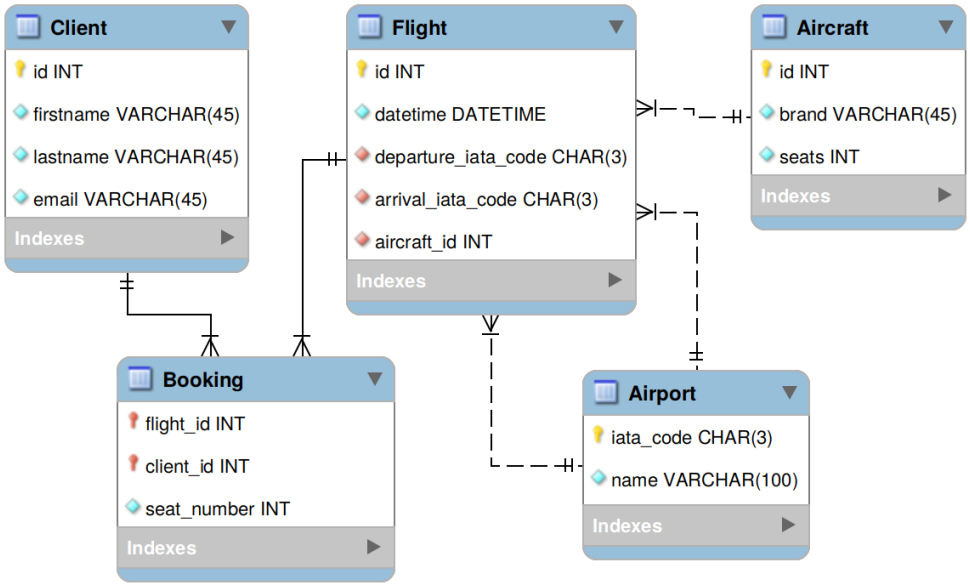



FIGURE 9 – Diagramme relationnel initial

### 3.1.2 Génération de données

Dans une démarche de réalisme et pour pouvoir faire des mesures de performances à plus grande échelle, des données initiales ont été générées. L'idée est de démarrer avec plusieurs milliers d'enregistrements cohérents, mais factices. Une partie de ces données a été générée à l'aide de Mockaroo [46], un générateur de données et APIs aléatoires. Les formats de sortie sont multiples (CSV, SQL, etc.), pour la plupart au format tabulaire. Les types des champs sont nombreux et au besoin personnalisables à l'aide du langage Ruby. 10'000 `clients` avec leurs noms, prénoms et emails ont pu être générés avec Mockaroo. Concernant les aéroports, une liste arbitraire d'une cinquantaine d'entrées est utilisée. En revanche la génération plausible des autres données est une partie plus difficile, du fait des contraintes sur les entités et associations suivantes :

- Un siège ne peut être attribué qu'à un seul client sur un vol
- Un client ne peut avoir qu'un seul siège sur un vol
- L'aéroport de destination d'un vol doit être différent de celui de départ
- Un vol ne peut offrir plus de places que le nombre maximal de places disponibles dans l'avion qui lui a été attribué

D'autres contraintes moins fortes sont désirées comme avoir des horaires "réalistes" pour les vols. Pour ces raisons et plutôt que générer une quantité énorme de tentatives de *bookings* et de *flights* aléatoires, puis par exemple de les insérer dans une base de données relationnelle

en se reposant sur ses contraintes d'intégrité, un programme Scala (`DataGenerator.scala`) est disponible pour générer des données initiales cohérentes. À partir d'une liste d'aéroports, de clients et de marques d'avions, ce programme génère les entités et associations restantes (à savoir les vols et les réservations). Ces listes d'entités peuvent ensuite être passées à une implémentation d'un `FlyManager` existant pour être persistées dans la base de données correspondant au `FlyManager` (voir sous-section 3.1.3 pour plus de détails).

### 3.1.3 Base commune

Pour que les différentes implémentations puissent être utilisées par un même simulateur/API et comparées entre elles sous plusieurs perspectives, un "contrat" d'implémentation a été établi, sous forme de trait Scala, le `FlyManager`, dont un extrait est disponible au listing 9. Il décrit le CRUD que doit offrir le système, les commandes (*create*, *update* et *delete*) et les requêtes (*read*). On remarque que ce modèle est synchrone, il est effectivement moins adapté à des *patterns* de *stream processing*, mais il est délibérément conçu ainsi illustrer la transition du modèle relationnel vers un modèle orienté événements. En héritant de ce trait, les différentes implémentations peuvent être utilisées comme source des données pour simulateur/API. Les types attendus et/ou retournés par les fonctions du `FlyManager` sont également communs à toutes les implémentations. Ils sont, pour la plupart, un *matching* des entités décrites dans le diagramme de la figure 8. Les variants "`History`" incluent en plus des champs de type `date` ou `timestamp` pour déterminer la date de l'état de l'attribut correspondant. Le listing 10 illustre ces différences.

```scala
trait FlyManager {
  // Commands
  def createAirport(a: Airport): Unit
  def createAircrafts(as: List[Aircraft]): Unit
  def updateBookingSeat(b: Booking, newSeat: Int): Unit
  def deleteFlight(f: Flight): Unit
  // Queries
  def client(id: Int): Option[(Client, Status)]
  def client(flightId: Int, seatNumber: Int): Option[(Client, Status)]
  def seatNumber(flightId: Int, clientId: Int): Option[(Int, Status)]
  def mostFlyableAircraftsBrands: Map[String, Int]
  def availableSeats(f: Flight): List[Int]
  def avgFlightOccupancy(f: Flight): Option[Double]
  def clientFlights(c: Client): List[Flight]
  def airportsWithMostArrivals: Map[Airport, Int]
  def clientHistory(clientId: Int): (List[ClientHistory], Status)
```

```
17    def clientsHistory: Map[Int, (List[ClientHistory], Status)]
18    // ...
19  }
```

Listing 9 – Extrait du trait `FlyManager`

```
1  case class Client(
2    id: Int,
3    firstname: String,
4    lastname: String,
5    email: String
6  )
7  case class ClientHistory(
8    id: Int,
9    firstname: String,
10   lastname: String,
11   email: String,
12   tsFirstname: LocalDateTime,
13   tsLastname: LocalDateTime,
14   tsEmail: LocalDateTime
15 )
```

Listing 10 – Différences entre le type `Client` et `ClientHistory`

Le traçage des changements est effectué sur les attributs suivants (avec le timestamp relevé à chaque modification) :

- Le prénom d'un client (`Client`, champ `firstname`)
- Le nom d'un client (`Client`, champ `lastname`)
- L'email d'un client (`Client`, champ `email`)
- L'heure de départ d'un vol (`Flight`, champ `datetime`)
- Le numéro du siège du client sur un vol (`Booking`, champ `seat_number`)
- L'état de la réservation d'un client (`Booking`, champ `state`)

Certaines *queries* (celles concernant des informations sujettes aux changements) donnent une information supplémentaire sur le statut du ou des enregistrements retournés à travers l'énumération `Status` visible au listing 11. De cette manière, un enregistrement est soit actuellement présent dans la base (`PRESENT`), soit était présent mais dorénavant supprimé (`DELETED`), soit supprimé et jamais présent en base (`NEVER_PRESENT`), soit inconnu

(`UNKNOWN`). Ce dernier cas est surtout présent pour les implémentations ne gérant pas l'historique d'informations supprimées (notamment la pure relationnelle, voir sous-section 4.1).

```
1 sealed trait Status
2 case object PRESENT extends Status
3 case object DELETED extends Status
4 case object NEVER_PRESENT extends Status
5 case object UNKNOWN extends Status
```

Listing 11 – Enumération `Status`

Le déploiement des différents services ou serveurs de base de données (MySQL, Kafka, etc.) a été fait au moyen de Docker Compose [47], l'orchestrateur basique de containers Docker. Docker Compose permet de facilement "*packager*" une application et de la déployer sur différents systèmes d'exploitation. En cours de développement, il est aussi utile et pratique de pouvoir rapidement créer et détruire un environnement complet pour avoir l'assurance de partir sur des bases saines et reproductibles.

## 3.2 Purement relationnelle

### 3.2.1 Concept

Avec cette implémentation, nous restons entièrement dans un monde relationnel. Pour garder la trace des changements, il faut adapter le modèle initial (vu en figure 9). Prenons le cas des changements sur les attributs d'un client pour illustrer les implications d'une telle adaptation. Une première idée serait d'ajouter un nouveau champ `timestamp` à la table `Client`, comme illustré sur la figure 10. La clé primaire de cette table serait maintenant composée de l'`id` du client et de ce `timestamp` (*id est* un client visible sur différents moments temporels). À chaque fois qu'une action de mise à jour d'un client est demandée, un nouvel enregistrement de client serait effectué, avec le `timestamp` courant et les champs voulus modifiés. Cette manière de faire est relativement simple, mais problématique sur un point : on ne sait pas quel champ a été modifié (et quand) d'un enregistrement à l'autre sans les comparer explicitement. Des erreurs pourraient survenir, par exemple une modification du nom non voulue lors de la mise à jour de l'email.

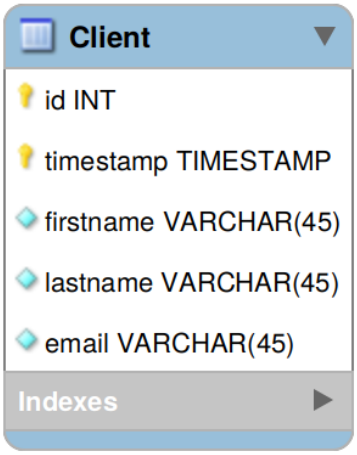



FIGURE 10 – Table `Client` avec champ `timestamp` supplémentaire

Une solution possible serait de rendre les champs nom, prénom et email optionnels (autorisation de valeurs `NULL`). Ainsi, lorsqu'on voudrait mettre à jour l'email d'un client par exemple, un nouvel enregistrement est effectué en mettant le nom et le prénom à `NULL`. Un exemple d'une telle séquence est visible à la table 3. On peut donc savoir le moment de modification par champs, mais le nouveau problème engendré par cette méthode, c'est que pour obtenir le client "entier" (avec tous ses attributs), il faut obligatoirement parcourir tous les enregistrements liés à ce client et prendre la dernière version en date de chacun pour le reconstituer.

| id | timestamp | firstname | lastname | email |
|---|---|---|---|---|
| 1 | 2021-01-01 08 :10 :15 | Johnny | Smith | jojo@mail.com |
| 1 | 2021-01-01 10 :56 :43 | `NULL` | `NULL` | jsmith@mail.com |
| 1 | 2021-01-01 11 :30 :32 | John | `NULL` | `NULL` |

TABLE 3 – Exemple de mises à jour d'un client

On retrouve ces mêmes problématiques pour les champs `datetime` de `Flight` et `state` de `Booking`. Une meilleure manière de faire (celle qui a été retenue) consiste à dépouiller la table "principale" de ses attributs pour ne garder que l'identifiant unique et autres attributs non sujets aux changements et dédier une table par attribut changeant (avec `timestamp`), avec comme clé étrangère l'identifiant de la table principale. La cardinalité est de un-à-plusieurs entre la table principale et les tables "attribut". Notre table `Client` se transforme donc en quatre tables : `Client`, `ClientFirstname`, `ClientLastname` et `ClientEmail`. Le traitement demeure certes plus lourd (il faut récupérer l'information de quatre tables au lieu d'une), mais on supprime les problèmes de valeurs nulles et de connaissance de quel champ modifié à quel moment. Les tables `Flight` et `Booking` suivent un raisonnement similaire pour leurs champs respectifs `datetime` et `state`. Le nouveau modèle produit sur cette logique est visible sous forme de diagramme à la figure 11.

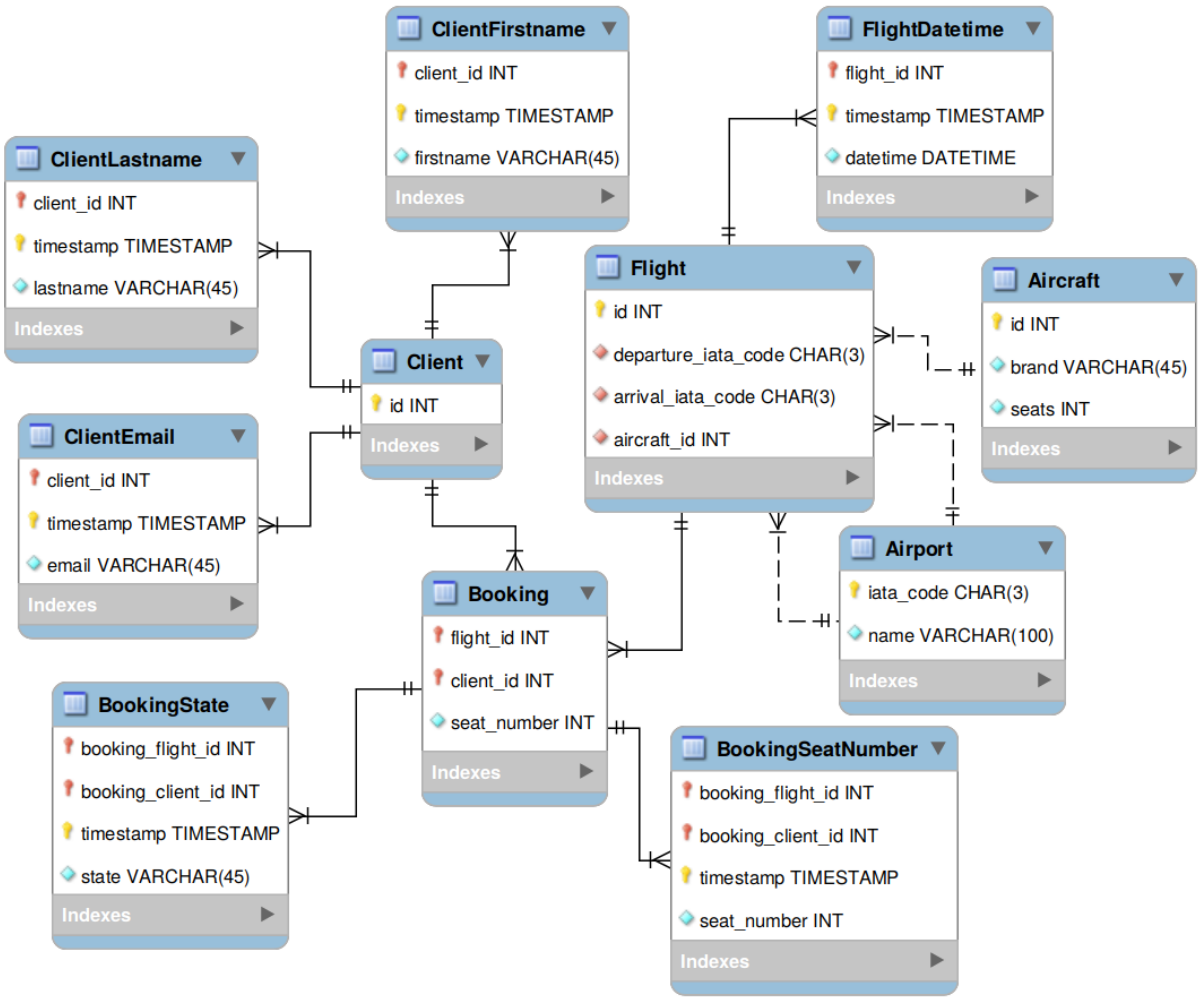



FIGURE 11 – Diagramme relationnel avec *tracking* des changements

Bien que ce modèle soit plus en adéquation avec les exigences de *tracking* de changements, il est tout de même soumis à quelques limitations :

- Pour garder la dépendance fonctionnelle $flight\_id, seat\_number \rightarrow client\_id$, *id est* le fait que sur un vol, un siège soit attribué à un client au plus, nous sommes obligés de garder ces trois attributs dans la table principale `Booking` et de les répéter dans la table `BookingSeatNumber`. On créé de ce fait une redondance d'information (`seat_number`) et un risque d'incohérence.
- Lors des actions de suppression (Booking, Client ou Flight), un choix a du être fait : soit toutes les occurences liées à l'identifiant de l'enregistrement sont supprimées (en perdant de ce fait l'historique, on supprime "réellement" de l'information), soit uniquement les enregistrements de la table principale sont supprimés, en mettant à `NULL` les champs correspondants dans les tables annexes (en perdant cette fois en intégrité, risque de réutilisation d'identifiants par exemple), soit on ne mets pas les clés principales comme clés étrangères dans les tables annexes, mais on perd cette fois les contraintes imposées, par exemple on ne peut plus être sûrs que l'enregistrement existe bel et bien. Une solution

intermédiaire pourrait être mise en place en ajoutant un statut à un enregistrement : existant ou supprimé. Mais, dans le cas d'un `Booking`, on empêche *de facto* un client d'annuler un vol puis de le reprendre après coup s'il change d'avis par exemple. La solution retenue est de supprimer purement et simplement l'enregistrement, en perdant l'historique définitivement.

### 3.2.2 Code

Cette implémentation, `MySQLFullFlyManager`, est un connecteur à une base de données relationnelle MySQL [2]. Elle se sert du connecteur JDBC [48] standard Java pour ouvrir une connexion à MySQL et exécuter les requêtes. La même connexion est utilisée pour chaque requête. Elle hérite, tout comme l'implémentation Debezium (voir sous-section 3.4), de la classe abstraite `MySQLFlyManager` avec qui elle partage la mécanisme de connexion, quelques fonctions concernant les aéroports et les avions (car les tables MySQL sont identiques) et trois fonctions, visibles au listing 12, qui sont présentes pour servir de base à pratiquement toutes les autres fonctions. Elles gèrent les mécanismes de connexion à MySQL, exécutent le *statement* donné en arguments et retournent au besoin le résultat. Dans le cas d'une commande, il n'y a pas de valeurs de retour. En cas de requêtes, on peut soit obtenir une `Option[T]`, soit une `List[T]`.

```
1  def command(s: String): Unit = try {
2    connection.createStatement.executeUpdate(s)
3  } catch {
4    case _: Throwable => ()
5  }
6  def queryToList[T](s: String, f: ResultSet => T): List[T] = {
7    def next(l: List[T], rs: ResultSet): List[T] = rs.next match {
8      case true  => f(rs) :: next(l, rs)
9      case false => l
10   }
11   try {
12     val rs = connection.createStatement.executeQuery(s)
13     next(Nil, rs)
14   } catch {
15     case _: Throwable => Nil
16   }
17 }
18 def query[T](s: String, f: ResultSet => T): Option[T] =
19   queryToList(s, f) match {
```

```
    case h :: _ => Some(h)
    case Nil    => None
  }
```

Listing 12 – Les trois fonctions de commandes et requêtes de `MySQLFlyManager`

Le gros du travail d'implémentation consiste à formuler les bonnes requêtes SQL. Le listing 13 montre quelques fonctions basiques de `MySQLFullFlyManager`. La plupart des requêtes sont relativement simples pour le développeur qui a l'habitude de servir de SQL. On peut néanmoins apercevoir sur ce listing 13 la limitation de cette implémentation quant à la suppression de l'historique : les fonctions retournant un `Status` indiquent uniquement `PRESENT` si l'information se trouve actuellement en base, sinon elles ne peuvent que retourner `UNKNOWN`, car cette information est perdue.

```
def createAirport(a: Airport): Unit = {
  command(
    s"""INSERT INTO Airport VALUES ("${a.iataCode}", "${a.name}")"""
  )
}
def seatNumber(flightId: Int, clientId: Int): Option[(Int, Status)] = {
  query(
    s"""
      SELECT seat_number FROM Booking
      WHERE flight_id = $flightId AND client_id = $clientId
    """,
    Utils.rsToInt
  ) match {
    case Some(v) => Some((v, PRESENT))
    case None    => None
  }
}
def totalSeatsNumber(f: Flight): Option[Int] = {
  query(s"""SELECT seats FROM Flight
      JOIN Aircraft ON Flight.aircraft_id = Aircraft.id
      WHERE Flight.id = ${f.id}""",
    Utils.rsToInt
  )
}
```

Listing 13 – Quelques fonctions basiques de `MySQLFullFlyManager`

Certaines fonctions, nécessitant de modifier plusieurs tables en même temps, sont exécutées au sein d'une transaction [49]. De cette manière, on s'assure que la séquence des `INSERT`, `UPDATE` et / ou `DELETE` se déroule de manière atomique. C'est notamment le cas d'une mise à jour d'un siège sur un vol, fonction visible au listing de code 14.

```
1  def updateBookingSeat(b: Booking, newSeat: Int): Unit =
2    try {
3      connection.setAutoCommit(false)
4      connection.createStatement.executeUpdate(
5        s"""UPDATE Booking SET seat_number = $newSeat
6          WHERE flight_id = ${b.flightId} AND client_id = ${b.clientId}"""
7      )
8      connection.createStatement.executeUpdate(
9        s"""INSERT INTO BookingSeatNumber VALUES
10         (${b.flightId}, ${b.clientId}, "${Utils.nowStr}", $newSeat)
11       """
12     )
13     connection.commit
14   }
15   catch { case e: Throwable => connection.rollback }
16   finally { connection.setAutoCommit(true) }
```

Listing 14 – Fonction `updateBookingSeat` nécessitant l'utilisation d'une transaction SQL

Les requêtes les plus complexes sont celles retournant une liste d'entités avec tous leurs attributs, avec uniquement le dernier état de ces derniers, par exemple la liste des clients avec prénom, nom et email les plus récents. Dans les tables `ClientFirstname`, `ClientLastname` et `ClientEmail`, il peut y avoir potentiellement plusieurs enregistrements correspondants à un même identifiant de client mais avec des valeurs de *timestamp* et d'attributs spécifiques différentes (le fameux dernier état). Lorsqu'on désire récupérer cette liste, il convient de regrouper les enregistrements par id de client, avec la clause SQL `GROUP BY`. Le problème est qu'on ne contrôle pas l'ordre du `GROUP BY` effectué, alors que nous aimerions être sûrs de prendre l'enregistrement correspondant au *timestamp* le plus récent. Nous aimerions donc écrire une requête ressemblant à celle disponible au listing 15. Mais cette requête est fausse, la clause `ORDER BY` doit être exécutée en dernier, après avoir groupé.

```
SELECT email FROM ClientEmail
ORDER BY timestamp DESC       # Interdit de faire un ORDER BY
GROUP BY client_id            # avant un GROUP BY
```

Listing 15 – Fausse requête SQL récupérant les emails des clients, groupés par client et triés par *timestamp* le plus récent

Une solution possible est décrite par Eddie Carrasco sur son blog [50]. Elle consiste en une sous requête (un `SELECT` imbriqué dans un autre) réordonnant les enregistrements selon les critères désirés, puis appliquant le `GROUP BY` et `ORDER BY` finaux, comme prévu. La sous-requête se sert de la fonction *window* [51], [52] (réalisant une opération d'aggrégation sur un ensemble de lignes), `OVER()` [53], assignant à chaque ligne un nombre, commençant à 1 et incrémenté à chaque ligne, comme une sorte d'identifiant virtuel. La requête finale pour récupérer la liste des clients, avec pour chaque client la dernière version des attributs nom, prénom et email insérés est disponible au listing 16. Cette technique est utilisée à plusieurs reprises pour les requêtes sur les tables avec traçage des changements.

```
SELECT id, firstname, lastname, email FROM (
  SELECT ROW_NUMBER() OVER (
    ORDER BY id,
      ClientFirstname.timestamp DESC,
      ClientLastname.timestamp DESC,
      ClientEmail.timestamp DESC
  ) AS virtual_id, id, firstname, lastname, email
  FROM Client AS C
  JOIN ClientFirstname ON C.id = ClientFirstname.client_id
  JOIN ClientLastname ON C.id = ClientLastname.client_id
  JOIN ClientEmail ON C.id = ClientEmail.client_id
) AS sorted_timestamps
GROUP BY sorted_timestamps.id
ORDER BY sorted_timestamps.id
```

Listing 16 – Requête SQL récupérant la liste des clients, avec pour chaque client la dernière version des attributs nom, prénom et email insérés

## 3.3 Orienté message broker avec Kafka

### 3.3.1 Concept

Cette implémentation est à l'opposé de la précédente (MySQL *full*), elle repose entièrement et uniquement sur Kafka pour la persistance des données. L'unité de données de base est ici l'événement ou le message. Chaque message envoyé dans un *topic* doit indiquer sa raison d'être, dans notre cas laquelle des opérations *create*, *update* ou *delete* il effectue, par exemple "Créer un client, du nom de Bob". Un *topic* par type d'entité est créé. Le format des données est ici choisi librement, étant donné que les messages sont produits et consommés par la même application. Les données sont donc organisées de manière assez primitive. L'avantage est que les opérations de lecture sont simples, il suffit de lire le contenu du *topic* concerné dans l'ordre chronologique pour obtenir l'historique des changements, tandis que le dernier état est présent dans le dernier message de la donnée concernée. L'écriture est également simple, un message avec une opération et une donnée est envoyé dans un *topic*, en revanche, c'est à la responsabilité du producteur de produire des données valides (voir section 4.6). Comme Kafka ingère les données sans format et contraintes en particulier et qu'il n'y a pas de moteur de requêtes comme dans MySQL, il faut entièrement calculer les résultats voulus depuis la logique métier, directement dans le code. On a entièrement le contrôle de ce qu'il se passe, mais il faut également "réinventer la roue".

### 3.3.2 Code

Ce tutoriel de Loïc Divad [19] a servi de guide de prise en main, il montre au développeur Scala comment déployer des programmes producteurs et consommateurs Kafka. Dans un soucis de simplicité, les clés et valeurs des événements Kafka ont comme type brut `String`. Chaque message est produit avec la syntaxe arbitraire présentée au listing 17. Dans cet exemple, nous avons cinq messages produits dans le *topic* des clients. Le premier élément est la clé primaire, ici l'id des clients. Le contenu proprement dit du message, sa valeur, vient ensuite, dans un pseudo format CSV avec comme séparateur le caractère dièse #. La première information de la valeur indique l'opération de ce message, pour rappel `c` pour *create*, `u` pour *update* et `d` pour *delete*. Aux lignes 1 et 2, deux nouveaux clients sont créés. À la ligne 3, l'email du client 1 est mis à jour (on note que tous les champs sont répétés, dans un souci de cohérence). La ligne 4, le client avec l'id 2 est supprimé.

```
1  c#1#Grover#MacQuarrie#gmacquarrie0@ezinearticles.com
2  c#2#Cornie#Chat#cchat1@sciencedaily.com
1  u#1#Grover#MacQuarrie#mynewmail@mail.com
2  d#2#Cornie#Chat#cchat1@sciencedaily.com
3  c#3#Ame#Jurczik#ajurczik2@bbc.co.uk
```

Listing 17 – Exemple de messages produits dans le *topic* des clients

Pour rester à l'écoute des derniers changements survenus, il faut maintenir un canal ouvert sur les *topics* correspondants. Car il ne serait pas performant de relire entièrement l'historique depuis Kafka au moment de retrouver une information. C'est pourquoi tout ce traitement est fait dans un *thread* séparé, `KafkaFlyManager` implémentant l'interface Java `Runnable`. Au lancement du *thread*, une connexion au *broker* Kafka est initialisée et l'entièreté des *topics* est lue (tête de lecture du *topic* placée au tout début). Chaque événement est ensuite traité de la manière suivante : tout d'abord, le *topic* du message est déterminé. Ensuite, la clé et la valeur du message sont désérialisées. Après, en fonction de l'opération réalisée sur l'enregistrement, un nouvel objet modélisant l'entité est produit. Ce dernier est ensuite placé dans une `Map[]` Scala correspondante, pour accès rapide futur dans les fonctions `query` et pour appliquer des contraintes d'existence. Le listing 18 illustre ce principe, en montrant le cas de gestion pour un client. En cas de mise à jour, le champs qui a changé est déterminé, pour ne mettre à jour que le timestamp associé et pas les autres.

```
while (true) {
  val records = consumer.poll(100)
  for (record <- records.asScala) {
    val maybeFlyEvent = readFrom(record)
    if (maybeFlyEvent.isDefined) {
      val (op, flyEvent) = maybeFlyEvent.get
      flyEvent match {
        case Client(id, f, l, e) =>
          op match {
            case Create => {
              val now = LocalDateTime.now
              val ch = ClientHistory(id, f, l, e, now, now, now)
              clientsMap.put(id, (Array(ch), PRESENT))
            }
            case Update => {
```

```
16                val now = LocalDateTime.now
17                val oldCh = clientsMap(id)._1.last
18                val tsFirstname =
19                  if (f != oldCh.firstname) now else oldCh.tsFirstname
20                val tsLastname =
21                  if (l != oldCh.lastname) now else oldCh.tsFirstname
22                val tsEmail = if (e != oldCh.email) now else
                  oldCh.tsFirstname
23                val newCh =
24                  ClientHistory(id, f, l, e, tsFirstname, tsLastname,
                    tsEmail)
25                clientsMap.put(id, (clientsMap(id)._1 :+ newCh, PRESENT))
26              }
27              case Delete => clientsMap.put(id, (clientsMap(id)._1,
                DELETED))
28            }
29          ...
30        }
31      }
32    }
33  }
```

Listing 18 – Boucle infinie de consommation des nouveaux événements dans `KafkaFlyManager`

Les fonctions de création, mise à jour et suppression sont en général simples aussi, le comportement global est illustré au listing 19. Pour les fonctions de création, il faut d'abord regarder dans la `Map[]` si l'id est encore libre (l'inverse pour les fonctions de mise à jour). Ensuite, on écrit la nouvelle donnée dans le bon *topic*.

```
1  private def create(fly: Fly, topic: String, inMap: Boolean): Unit =
2    if (inMap) writeTo(topic, Create, fly)
3
4  override def createAirport(a: Airport): Unit =
5    create(a, airportTopic, !airportsMap.contains(a.iataCode))
6
7  private def update(newFly: Fly, topic: String, inMap: Boolean): Unit =
8    if (inMap) writeTo(topic, Update, newFly)
9
```

```scala
override def updateClientFirstname(c: Client, newF: String): Unit = {
  update(
    Client(c.id, newF, c.lastname, c.email),
    clientTopic,
    clientsMap.contains(c.id)
  )
}
```

Listing 19 – Fonctions créant et mettant à jour les clients dans `KafkaFlyManager`

La simplicité se trouve également du côté des fonctions retournant des entités "simples", leur dernier état ou leur historique. Au listing 20 on peut voir les fonctions récupérant un client. Contrairement à l'implémentation MySQL *full*, cette implémentation peut déterminer si un enregistrement a été supprimé, car son historique demeure dans le *topic*.

```scala
override def client(id: Int): Option[(Client, Status)] =
  clientsMap.get(id) match {
    case Some((clients, status)) =>
      Some((clients.last.toClient, status))
    case None => None
  }

override def clientHistory(
  clientId: Int
): (List[ClientHistory], Status) =
  clientsMap.get(clientId) match {
    case Some((chs, status)) => (chs.toList, status)
    case None                => (Nil, NEVER_PRESENT)
  }
```

Listing 20 – Fonctions récupérant un client et son historique dans `KafkaFlyManager`

En revanche, les fonctions qui se reposent lourdement sur les avantages de SQL, notamment lorsqu'il faut faire des `JOIN` entre tables et des `GROUP BY` sont plus complexes à mettre en oeuvre. Heureusement, les fonctions inclues avec Scala sur les collections sont suffisamment puissantes et élégantes pour reproduire le comportement voulu. Le listing 21 montre deux fonctions de ce genre.

```scala
override def mostFlyableAircraftsBrands: Map[String, Int] =
  flightsMap
    .filter { case (_, v) => v._2 == PRESENT }
    .map { case (k, v) =>
      (k, aircraftsMap(v._1.last.aircraftId).brand)
    }
    .values
    .groupBy(b => b)
    .map { case (brand, v) => (brand, v.toList.length) }

override def availableSeats(f: Flight): List[Int] =
  totalSeatsNumber(f) match {
    case Some(totalSeats) => {
      val takenSeats = bookingsMap
        .filter { case ((fId, _), (_, status)) =>
          fId == f.id && status == PRESENT
        }
        .values
        .map { case (bhs, _) => bhs.last.seat }
        .toSet
      val allSeats = (1 to totalSeats).toSet
      (allSeats diff takenSeats).toList.sorted
    }
    case None => Nil
  }
```

Listing 21 – Fonctions plus complexes, notamment sur les avions, de `KafkaFlyManager`

Globalement, cette implémentation a du potentiel, mais souffre de défauts majeurs : il n'y a pas de gestion de la concurrence des insertions des données, elle nécessite un mécanisme de transactions pour pouvoir être correcte.

## 3.4 MySQL + Debezium : changements kafkaïens

### 3.4.1 Concept

Avec cette implémentation, nous sommes à cheval entre le monde relationnel et une source de données purement orientée événements / messages. Pour rappel, on connecte ici notre base de données MySQL à Debezium (voir section 2.7), qui va monitorer les opérations `INSERT`, `UPDATE` et `DELETE` de chaque table et produire un événement associé dans un *topic* Kafka

pour la table en question. Grâce à Debezium, on garde les avantages d'une base de données relationnelle, avec ses contraintes et vérifications (notre "source de vérité" demeure fiable) et en même temps on accède par des *topics* Kafka à l'historique des changements. On applique ici en partie le *pattern* CQRS, avec le côté "commandes" entièrement maintenu par la base de données relationnelle et le côté requêtes avec historique servi en lisant dans les *topics* concernés. Le modèle relationnel utilisé ici est celui présenté initialement dans la sous-section 3.1.1 à la figure 9. Étant donné que la logique de gestion des changements est déportée dans Kafka, le modèle relationnel redevient "simple", on peut donc réutiliser le modèle initial. Lorsqu'un enregistrement est manipulé (opération `INSERT`, `UPDATE` ou `DELETE`), on va simplement exécuter la requête SQL prévue à cet effet sur la table concernée. On a donc en permanence la dernière version de chaque donnée en base de données (même pour une suppression, la donnée n'existe plus dans la base). En revanche, les changements d'états des informations sont lus en consommant les événements des *topics* Kafka. Ainsi, on gomme les défauts de l'implémentation purement relationnelle tout en gardant les contraintes offertes par MySQL.

### 3.4.2 Code

La prise en main de Debezium est aisée, le tutoriel sur le site officiel [54] est bien fait et prend l'utilisateur par la main. Grâce aux différents exemples fournis sur le Github du projet [55], un déploiement Docker Compose avec Kafka comme *message broker*, MySQL comme base de données relationnelle "à surveiller" et Debezium, par son image Docker `Connect` publiant les changements survenus dans MySQL dans Kafka, a pu être mis sur pied. La version de Debezium utilisée ici est la 1.5, dernière version stable en date. Étant donné que Debezium ne gère pas la partie consommation des événements (il se "contente" de publier dans le *topic* approprié ), la gestion de la connexion à Kafka et création d'un Kafka *consumer* est à notre charge, comme pour l'implémentation Kafka *full* (voir sous-section 3.3). Toutes les fonctions intéragissant uniquement avec MySQL sont synchrones, elles réutilisent la connexion partagée à MySQL ouverte à la construction d'un objet de type `MySQLDebeziumFlyManager`. En revanche, en ce qui concerne les historiques, la même stratégie que pour l'implémentation Kafka *full* a été mise en place, un *thread* écoute sur les changements provenant de Kafka. Seuls les *topics* `Booking`, `Client` et `Flight` sont lus, car pour les `Aircraft` et `Airport`, les données ne changent pas et peuvent être lues depuis MySQL. La désérialisation et le *parsing* des clés et valeurs en JSON sont effectués avec la librairie Scala spray-json [56].

Les fonctions de base visibles au listing 13 sont réutilisées telles quelles pour les commandes et requêtes intéragissant uniquement avec MySQL. Par contre, la fonction de base `query` est étendue pour les fonctions qui retourne un `Status` ou un historique : après avoir regardé dans MySQL et éventuellement retrouvé une information, on regarde si un historique d'événements n'est pas présent dans la `Map[]` à travers les fonctions `lookInTopic()` correspondante. Le code en question est visible au listing 22.

```scala
private def query[T, U](
    s: String,
    f: ResultSet => T,
    lookInTopic: U => Option[T],
    arg: U
): Option[(T, Status)] =
  super.query(s, f) match {
    case Some(v) => Some(v, PRESENT)
    case None => {
      val inTopic = lookInTopic(arg)
      if (inTopic.isDefined) Some(inTopic.get, DELETED)
      else None
    }
  }
```

Listing 22 – Redéfinition de la fonction query dans MySQLDebeziumFlyManager

Grâce à cette nouvelle base, une fonction comme seatNumber indique maintenant si un événement était présent en base de données par le passé et retourne la dernière version d'une réservation, contrairement à la version MySQL visible au listing 13. Le listing 23 illustre l'usage de la nouvelle fonction query pour récupérer la dernière version d'une réservation.

```scala
val bookingsMap = mutable.Map[(Int, Int), Array[BookingHistory]]()

def lookInBookingTopic(flightIdclientId: (Int, Int)): Option[Int] =
  bookingsMap.get(flightIdclientId)
    .flatMap(bs => if (bs.nonEmpty) Some(bs.last.seat) else None)

def seatNumber(flightId: Int, clientId: Int): Option[(Int, Status)] =
  query(s"""SELECT seat_number FROM Booking
      WHERE flight_id = $flightId AND client_id = $clientId""",
    Utils.rsToInt,
    lookInBookingTopic,
    (flightId, clientId)
  )
```

Listing 23 – Illustration de la fonction seatNumber de MySQLDebeziumFlyManager

Concernant les fonctions retournant un historique (BookingHistory, ClientHistory ou

`FlightHistory`), on procède de la manière suivante : on regarde dans MySQL uniquement pour indiquer si l'entité recherchée n'est pas supprimée, ensuite on récupère tout simplement la liste des versions de cette entité dans la `Map[]` correspondante. Le listing 24 illustre ce fonctionnement sur la fonction `clientHistory`.

```
1  def clientHistory(clientId: Int): (List[ClientHistory], Status) =
2    clientsMap.get(clientId) match {
3      case Some(cs) =>
4        if (cs.nonEmpty) {
5          val actualClient = query(
6            s"SELECT * FROM Client WHERE id = $clientId",
7            Utils.rsToClient
8          )
9          actualClient match {
10           case Some(_) => (cs.toList, PRESENT)
11           case None    => (cs.toList, DELETED)
12         }
13       } else (Nil, NEVER_PRESENT)
14     case None => (Nil, NEVER_PRESENT)
15   }
```

Listing 24 – Fonction récupérant l'historique d'un client dans `MySQLDebeziumFlyManager`

## 3.5 ksqlDB : historique avec *streams* et vues matérialisées

Cette sous-section est légèrement mensongère, dans le sens où il n'y a pas eu d'implémentation Scala avec ksqlDB à proprement parler. Il est vrai que ksqlDB offre un client Java (utilisable depuis Scala) et une API REST, mais en étudiant l'architecture de ksqlDB [57], on constate ksqlDB et Kafka Streams sont interchangeables dans la plupart des situations, le premier étant une abstraction de niveau plus élevé que le second (pour rappel, ksqlDB est construit par dessus Kafka Streams). Il semble donc préférable, du point de vue du programmeur, de se servir des Kafka Streams qui sont plus flexibles que ksqlDB. C'est pour cela que ksqlDB a été mis en pratique grâce à sa CLI uniquement, sur un jeu réduit de données de notre cas d'étude, pour illustrer ses fonctionnalités.

On démarre avec la *stack* minimale de développement de ksqlDB, à savoir un serveur Kafka (`broker`), le serveur ksqlDB (`ksqldb-server`) et la CLI de ksqlDB (`ksqldb-cli`). Le `ksqldb-server` place les données persistantes dans le `broker` et la `ksqldb-cli` se connecte au `ksqldb-server`. Les concepteurs de ksqlDB mettent à disposition des images Docker

pour la CLI et le serveur, il est donc aisé de déployer tout l'environnement à l'aide de Docker Compose [47] pour tester et développer. Nous obtenons ainsi une variation de l'implémentation avec Kafka uniquement (sous-section 3.3), sans base de données relationnelle, mais avec la surcouche ksqlDB. On commence par démarrer la `ksqldb-cli` avec la commande visible au listing 25. On peut remarquer qu'on passe un fichier de configuration initiale `ksql-init.conf` à la CLI. Nous y plaçons le contenu visible au listing 26. La première ligne indique que les requêtes sur les *streams* et tables doivent se faire depuis le début de l'historique (par exemple si on demande la liste entière des vols) et la deuxième fait en sorte que le serveur ksqlDB émette les enregistrements correspondants à une requête aussitôt reçus / calculés (zéro latence).

```
docker exec -it ksqldb-cli ksql --config-file /ksql-init.conf \
  -- http://ksqldb-server:8088
```

Listing 25 – Connexion à la CLI de ksqlDB

```
ksql.streams.auto.offset.reset = earliest
ksql.streams.cache.max.bytes.buffering = 0
```

Listing 26 – Fichier de configuration initiale de ksqlDB

La stratégie est de créer un *stream* et une table par entité (`client, flight, booking`, etc.), de sorte à ce que l'insertion / modification / suppression soit faite dans les *streams* (avec indication de l'opération dans le message) et que les lectures soient réalisées depuis les *streams* ou les tables, en fonction des besoins. Le listing 27 montre les instructions de création pour l'entité `client` et deux exemples d'insertions dans le *stream* `clients`. Le *stream* `clients` et la table `client` reposent tous deux sur le *topic* Kafka `clients`, créé pour l'occasion avec la clause `WITH` si non existant. Le format des messages des *topics* choisi ici est le JSON. On peut également voir que pour le *stream* et la table, l'attribut `id` est défini comme clé primaire. Il faut toutefois souligner que dans ces quelques manipulations, cette clé primaire était définie "à la main" lors des opérations d'insertions. ksqlDB n'offre pas de concept d'id auto-incrémenté comme on le trouve dans MySQL. C'est à la charge du développeur de gérer cette logique en amont, car les erreurs peuvent vite être commises : en effet, un enregistrement, étiqueté comme nouveau, qui reprendrait par erreur l'id d'un autre "annulerait" l'historique de ce dernier.

```
CREATE STREAM clients (
  id INT KEY, firstname STRING, lastname STRING,
  email STRING, op STRING
) WITH (kafka_topic='clients', format = 'json', partitions = 1);
CREATE TABLE client (
  id INT PRIMARY KEY, firstname STRING, lastname STRING, email STRING
) WITH (kafka_topic='clients', format = 'json');

INSERT INTO clients VALUES (1, 'Bob', 'Smith', 'bs@mail.com', 'c');
INSERT INTO clients VALUES (2, 'Fred', 'Chat', 'fc@mail.com', 'c');
...
```

Listing 27 – Création du *stream* et de la table pour l'entité `Client` dans ksqlDB

Ainsi, selon si on désire récupérer l'état courant d'un enregistrement ou son historique, on peut puiser l'information depuis le *stream* ou la table correspondant. Pour essayer les spécificités de ksqlDB, plusieurs requêtes *push* tirant profit de ksqlDB, hors contrat défini au préalable (voir sous-section 3.1.3) peuvent être formulées. Elles sont toutes vouées à réaliser la détection de changements :

- Détecter quel client change de siège un jour avant départ, utile au marketing, qui pourrait par exemple offrir des options bonus pour le nouveau siège choisi.
- Alerter en cas de "fraude" : des changements d'attributs sur un court laps de temps (par exemple des changements de siège intempestifs).
- Visualiser le remplissage de l'avion en "temps réel" : lorsqu'un vol est plein à 90% par exemple, ouvrir des places à prix bradés, lancer des procédures administratives, etc.
- Envoyer un email au client lorsqu'une de ses réservations change d'état.

Le listing 28 montre les deux premières requêtes de la liste ci-dessus. Dans la première, les réservations (*bookings*) ainsi retournées contiennent les identifiants des clients concernés. Dans la deuxième, on peut voir l'utilisation du fenêtrage de ksqlDB, avec la clause `WINDOW`. Dans les deux cas, en cas d'insertions d'événements correspondant aux critères, on peut voir les résultats être mis à jour en temps réel.

```
-- Check if bookings are update 1 day before departure
SELECT *, (UNIX_TIMESTAMP(flight.departure) - UNIX_TIMESTAMP()) AS diff
FROM bookings JOIN flight ON bookings.flight_id = flight.id
WHERE
  bookings.op = 'u' AND
  (UNIX_TIMESTAMP(flight.departure) - UNIX_TIMESTAMP()) <=
  (1000 * 3600 * 24) -- milliseconds in one day
EMIT CHANGES;

-- Check if 3 or more changes in seat on a 30 seconds window occurs
SELECT
  flight_id, client_id, latest_by_offset(seat_number) AS seat,
  count(*) AS c, WINDOWSTART, WINDOWEND
FROM bookings
WINDOW TUMBLING (SIZE 30 SECONDS, RETENTION 1 DAYS)
GROUP BY flight_id, client_id
HAVING count(*) >= 3
EMIT CHANGES;
```

Listing 28 – Exemple de requêtes ksqlDB *push* et avec fenêtrage temporel

## 3.6 MySQL + Debezium + Kafka Streams

Cette implémentation (`MySQLDebeziumKStreamsFlyManager`) est une sorte de "patchwork" des autres, elle repose en effet sur celle avec MySQL + Debezium (sous-section 3.4) en se servant en plus des Kafka Streams (voir sous-section 2.5.6), qui reprennent les idées et concepts de ksqlDB (sous-section 3.5). En conséquences, toute la partie commandes est héritée de `MySQLDebeziumFlyManager`. La partie désérialisation a également pu être reprise (`FlyDebeziumDeserializer`). Seule la partie requêtes est différente, utilisant Kafka Streams. La stratégie est ici de se servir des *streams* (`KStream`) et des tables (`KTable`) pour remplacer les requêtes en lecture faites à MySQL et à l'usage des `Map[]` côté Scala. De cette manière, en théorie du moins, on applique le *pattern* CQRS complètement, en utilisant MySQL uniquement pour le côté écriture et délégant la partie lecture aux Kafka Streams. La première étape consiste à épurer les messages des *topics* Debezium, pour ne garder que l'essentiel. Le listing 29 montre le processus pour le *topic* concernant les clients. En tirant profit du mécanisme de la dualité *streams* / tables (voir table 2), on définit un nouveau *topic* `client-stream` traduisant les événements provenant de Debezium et une table associée, représentant l'état

courant d'un enregistrement. Cette table est matérialisée dans le *topic* `client-table`. Chaque message de ces *topics* a la structure visible sur le listing 30, avec la clé au format brut (ici en `Int`) et les données en JSON, toujours manipulées avec spray-json.

```
val dbzClients: KStream[String, String] =
  builder.stream[String, String]("dbserver.fly.Client")
val clientStream: KStream[Int, String] = dbzClients.map((k, v) => {
  val event = eventToFlyModelEvent(k.parseJson, v.parseJson)
  if (event.isDefined) {
    event.get match {
      case a: ClientEvent => {
        val client = a.op match {
          case Create | Update => a.after.get
          case Delete          => a.before.get
        }
        (client.id, client.toJson.toString)
      }
    }
  }
})
clientStream.to("client-stream")

val clientTable: KTable[Int, String] = builder.table("client-stream")
clientTable.toStream.to("client-table")
```

Listing 29 – Conversion des *topics* Debezium dans `MySQLDebeziumKStreamsFlyManager`

```
1, {"email":"bob@mail.com","firstname":"Bob","id":1,"lastname":"Smith"}
```

Listing 30 – Exemple de message produit dans le *stream* `client-stream`

Les fonctions de requêtes se résument ainsi à regarder dans le *stream* lorsque un historique est demandé, ou de regarder dans les tables si c'est uniquement le dernier état qui nous intéresse. Le listing 31 donne un exemple pour la fonction retournant un client à l'aide de son id. En jouant sur la présence ou non de données dans le *stream* et la table, on peut calculer le `Status` de l'enregistrement. Malheureusement, cette démonstration de code est théorique, car pour une raison inconnue, l'exécution du code n'effectue pas le travail attendu.

```scala
override def client(id: Int): Option[(Client, Status)] = {
  val builder = new StreamsBuilder
  var clientStates = Array[Client]()
  builder
    .stream[Int, String]("client-stream")
    .filter((k, _) => k == id)
    .foreach((k, v) => {
      val c = v.parseJson.convertTo[Client]
      clientStates = clientStates :+ c
    })
  var present = false
  builder
    .table[Int, String]("client-table")
    .filter((k, _) => k == id)
    .toStream
    .foreach((k, v) => present = true)

  (clientStates.nonEmpty, present) match {
    case (true, true) => Some((clientStates.last, PRESENT))
    case (true, false) => Some((clientStates.last, DELETED))
    case (false, false) => Some((clientStates.last, NEVER_PRESENT))
    case _ => None
  }
}
```

Listing 31 – Fonction `client()` de `MySQLDebeziumKStreamsFlyManager`

# 4 Discussions et comparaisons des implémentations

Dans cette section, nous allons résumer les avantages et inconvénients de chaque implémentation. Ensuite, une synthèse sera effectuée pour déterminer lesquelles sont le mieux adaptées pour notre situation.

## 4.1 Purement relationnelle

### 4.1.1 Avantages

- Cohérence des données : avec les contraintes d'intégrité offertes par une base de données relationnelle, il est plus facile de réagir à des données non conformes. Les données écrites peuvent être considérées comme de "confiance".
- Les transactions (ACID) lors de modifications apportées aux données confèrent une assurance supplémentaire et un mécanisme fiable de gestion de la concurrence.
- Du fait de leur modèle inhérent, les bases de données relationnelles réduisent la redondance des données, en forçant la séparation des entités.
- Cette implémentation ne se sert que d'un seul type de base de données, MySQL, il n'y a donc qu'une seule technologie à maîtriser.

### 4.1.2 Inconvénients

- L'historique des changements n'est pas entièrement préservé : on constate dans notre cas que, si un enregistrement est supprimé, on perd son historique, on peut monitorer uniquement les mises à jour.
- Complexité des requêtes : on réalise que même pour notre cas de figure relativement peu complexe en nombre de tables, associations et contraintes, les requêtes à réaliser en lecture deviennent rapidement longues et complexes.
- Scalabilité : les bases de données relationnelles sont scalables de manière verticale (augmentation de la puissance de l'unique machine), elles sont difficilement scalables de manière horizontale (ajout de machines).
- La sémantique des bases de données relationnelles est peu adaptée à l'*event sourcing*, elle est pensée en termes de CRUD classique.

## 4.2 Kafka uniquement

### 4.2.1 Avantages

- La partie Commandes (*create, update, delete*) est très rapide (check dans les `Map[] +` envoi de message).

- Les lectures (et relectures) de l'historique sont rapides, c'est une séquence continue de messages.
- Une seule technologie à maîtriser, Kafka.
- Pas de surcouche, on a le contrôle total du comportement du programme, car la couche de la base de données est fine.
- Kafka est horizontalement scalable et distribué, il s'adapte très bien à une forte montée en charge (en théorie, voir dernier désavantage).
- Cette implémentation est purement orientée événements : elle est adaptée pour un historique de changements.
- Grâce aux mécanismes inhérents à Kafka (partitionnement), les données peuvent facilement être redondantes.

### 4.2.2 Inconvénients

- En contrepartie du contrôle total, il n'y a aucune "facilité", cette implémentation est presque trop brute, c'est au développeur de tout (ré)implémenter.
- Les contraintes lors des commandes doivent être gérées entièrement côté logique métier, dans le code.
- La gestion des clés primaires des entités est héritée du modèle relationnel : on ne peut l'appliquer au modèle Kafka en l'état.
- Pas de transactions en l'état, il faut se servir des transactions Kafka, voir la synthèse (sous-section 4.6).
- Pas de langage de requêtes, liste de messages brute, pas de jointures.
- Le goulet d'étranglement vient du code applicatif : s'il n'est pas parallèle et / ou distribué (comme dans notre cas simple ici présent), la montée en charge devient difficilement tenable.

## 4.3 MySQL + Debezium

### 4.3.1 Avantages

- On garde une source de données cohérente et fiable avec MySQL gérant l'écriture.
- La lecture des dernières version de données est facile.
- Debezium est facilement configurable par dessus une *stack* Kafka et MySQL existante.
- Les requêtes SQL restent simples, car les tables le sont aussi.
- L'historique des changements est sauvegardé durablement dans Kafka.
- Une trace des enregistrements supprimés est maintenue dans Kafka.
- Application partielle du pattern CQRS.

### 4.3.2 Inconvénients

- Dépendance supplémentaire à Debezium.
- Il y a un *trade-off* entre les performances et la mémoire vive occupée : en plaçant l'historique des changements dans des `Map[]` Scala, on a certes l'accès à l'historique très rapidement, mais en cas de grandes quantités de données, on risque de saturer la mémoire vive (généralement plus rapidement que le disque dur).
- Du fait de la composition MySQL + Kafka, cette implémentation hérite de certains défauts des deux, comme la scalabilité en partie verticale, certaines requêtes (historique) doivent être construites un peu "à la main" (*id est* il n'y a pas de moteur de requêtes pour cela).

## 4.4 ksqlDB only

### 4.4.1 Avantages

- Son pseudo SQL de requêtes est suffisant dans de nombreux cas de lecture.
- Grâce aux *streams*, un flux d'événements peut être transformé facilement en un autre.
- Les *push queries* permettent de mettre sur pied des systèmes de notifications rapidement.
- ksqlDB dispose de nombreux connecteurs : MySQL, Redis, Elasticsearch, MongoDB, etc.
- Les vues matérialisées avec fenêtrage (*windowing*) sont intéressantes en cas de traitement sur une période de temps définie.
- ksqlDB est pratique si on ne veut pas programmer directement, que ce soit pour des *proof of concept* rapides our pour des applications plus lourdes.

### 4.4.2 Inconvénients

- Pas aussi contraignant qu'une base de données relationnelle (intégrité et transactions).
- Comme pour Kafka, le format des clés primaires doit être adapté : les contraintes doivent être posées ailleurs.
- Couche supplémentaire à Kafka, devant être comprise et pouvant porter à confusion sur certains principes.

## 4.5 MySQL + Debezium + Kafka Streams

### 4.5.1 Avantages

- Patchwork des avantages des technologies qui la compose :
  - Contraintes d'intégrité (MySQL)

    - Cohérence des données (MySQL)
    - Gestion de l'historique
    - Orienté événements
- Les Kafka Streams sont plus flexibles que ksqlDB.
- La dualité entre tables et *streams* permet de se passer de MySQL en lecture (en théorie).

### 4.5.2 Inconvénients

- Même si les fonctions offertes par Kafka Streams sont puissantes, elles ne remplacent pas totalement les possibilités des collections Scala "classiques", cette implémentation est moins flexible que celle MySQL + Debezium sur ce point.
- Les désavantages de ksqlDB se retrouvent ici : surcouche à Kafka, pas aussi contraignant, etc.

## 4.6 Récapitulatif et synthèse

La figure 12 résume en quelque sorte le positionnement de chaque implémentation. Du côté du monde relationnel se trouve l'implémentation MySQL *full*. Dans notre cas d'études, elle est à écarter car elle ne respecte pas la contrainte de maintenir l'historique en cas de suppression d'enregistrements. À l'extrême inverse, du côté des événements, se trouve l'implémentation Kafka *full*. Elle comporte elle aussi des défauts, notamment son manque de cohérence et de transactions en l'état. Ces défauts peuvent être gommés de plusieurs manières. De prime abord, Kafka seul semble ne pas respecter le concept ACID, mais en regardant de plus près, il l'applique pourtant. Grâce à ses messages immutables, la lecture et écriture des événements sont atomiques. Les données sont durables, car écrites sur le disque. Chaque événement est unique, il est isolé d'un autre. Plusieurs possibilités existent pour garantir la consistence et la cohérence : la politique du message "*exactly-once-consumed*", expliquée par Ben Stopford [58], en rendant idempotents les messages envoyés (a le même effet qu'ils soient envoyés une ou plusieurs fois). Il ne faut pas qu'un message dupliqué (par erreur) puisse casser la logique métier. Ensuite, la gestion des clé primaires doit être différente que dans MySQL. Enfin, les transactions Kafka (concept illustré par Michael Seifert [59] et détaillé par Apurva Mehta et Jason Gustafson [60]) semblent être la solution finale pour garantir la cohérence des données. Par ces arguments, ainsi que les synthèses de Neil Avery [61] et Kevin Petrie [62], on peut conclure que Kafka peut garantir les transactions ACID.

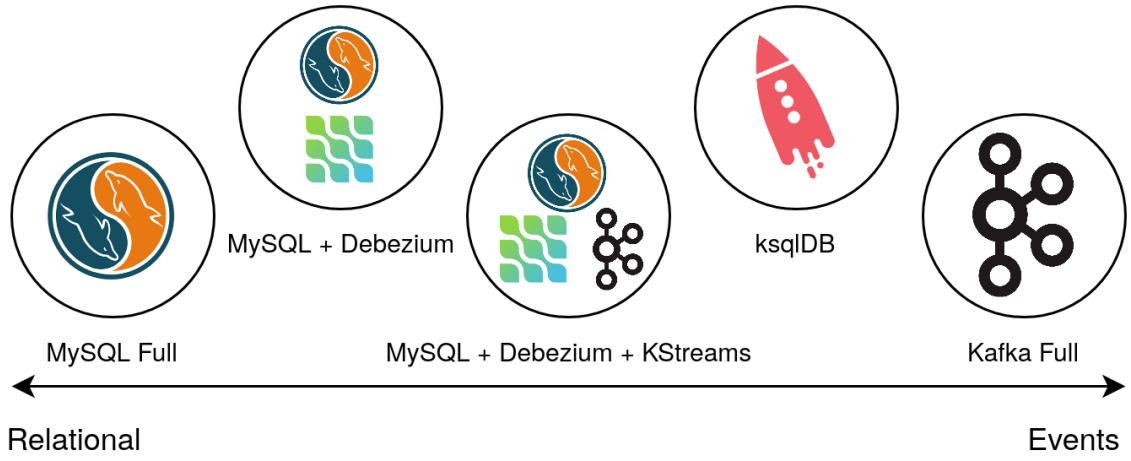



FIGURE 12 – Spectre des implémentations, triées en fonction du *pattern*

Comme dit Kai Waehner [63], Kafka et son écosystème peuvent être utilisés seuls en tant que source de données de confiance, mais ne remplacent pas des bases de données comme MySQL, MongoDB, Elasticsearch ou Hadoop. En remettant le focus sur notre cas d'étude, quelle stratégie est la plus simple pour migrer une base relationnelle vers une base de données orientée événements ? Les solutions hybrides à base de MySQL, Debezium et / ou ksqlDB semblent être la meilleure piste. En effet, les contraintes fortes d'intégrité des bases de données relationnelles sont bien pratiques, il est dommage de les perdre. Si on veut transformer une solution relationnelle vers du "tout Kafka", il faut penser le système en terme d'événements dès le départ, ne pas raisonner comme avec les bases de données relationnelles.

# 5 Conclusion

## 5.1 Bilan

Les objectifs principaux de ce projet étaient d'étudier différentes solutions de bases de données orientées événements, en les comparant au modèle relationnel en choisissant un *use case* afin d'en réaliser un *proof of concept*. Il en ressort que chaque implémentation a ses avantages et inconvénients. Elles ne sont pas toutes totalement adéquates pour notre cas d'étude, à l'exception de celles mixant MySQL, Debezium et / ou Kafka Streams. Elles détiennent un grand nombre d'avantages : implémentation "complète" (pas comme MySQL *full*), cohérence des données en écriture (implémentation partielle ou totale de CQRS), relecture efficace de l'historique dans Kafka, connexion "gratuite" entre MySQL et Kafka grâce à Debezium, etc. Leur défaut majeur est la multiplication des technologies à maîtriser. Ces implémentations renforcent le *pattern event sourcing* tout en véhiculant des concepts liés aux microservices.

## 5.2 Problèmes rencontrés

Aucun problème majeur n'a été rencontré, sauf pour la prise en main des Kafka Streams, qui a débouché sur la non-implémentation complète de `MySQLDebeziumKStreamsFlyManager`. Quelques difficultés sont apparues au moment de faire la synthèse entre les différentes couches Kafka. En effet, les mêmes notions se répètent au sein de Kafka, Kafka Streams et ksqlDB, mais avec des différences subtiles.

## 5.3 Améliorations possibles

Bien que la réalisation de ce travail ait permis d'approfondir certaines technologies déjà connues, il est toujours possible de creuser le sujet. Voici une liste non exhausive des améliorations potentielles :

- Dans un souci de simplicité pour le *proof of concept*, toutes les fonctions du contrat des implémentations sont synchrones. Il serait intéressant de les convertir en asynchrone, pour tirer profit du matériel multi-coeurs et des structures telles que les *streams*.
- La réalisation d'un "vrai" simulateur, ou d'une API utilisant les implémentations, aurait permis de comparer les implémentations sur l'aspect des performances (temps d'exécution).
- Assez rapidement, des alternatives à Kafka ont été écartées, même si certaines, comme ApacheMQ auraient pu être utilisées.
- Approfondir les fonctionnalités de Kafka, comme les transactions, ou appliquer certains *patterns* de systèmes distribués, qui seraient équivalents aux contraintes d'intégrité des bases de données relationnelles, pour définitivement se passer d'elles.

# 6 Références


[1] Joël Cavat. Systèmes de base de données. https://gitedu.hesge.ch/joel.cavat/sbd2020, juin 2020. Consulté en février 2021.

[2] Michael Widenius. Mysql. https://www.mysql.com/fr/. Consulté en mars 2021.

[3] MariaDB Foundation. Mariadb. https://mariadb.org/. Consulté en mars 2021.

[4] Michael Stonebraker. Postgresql : The world's most advanced open source relational database. https://www.postgresql.org/. Consulté en mars 2021.

[5] Microsoft. Microsoft data platform. https://www.microsoft.com/fr-ch/sql-server. Consulté en mars 2021.

[6] Wikipédia. Change data capture. https://en.wikipedia.org/wiki/Change_data_capture. Consulté en mars 2021.

[7] Clément Héliou. Event sourcing : comprendre les bases dun système évènementiel. https://blog.engineering.publicissapient.fr/2017/01/16/event-sourcing-comprendre-les-bases-dun-systeme-evenementiel/, janvier 2017. Consulté en mars 2021.

[8] Microsoft Docs. Event sourcing pattern. https://docs.microsoft.com/en-us/azure/architecture/patterns/event-sourcing, juin 2017. Consulté en mars 2021.

[9] Martin Fowler. Event sourcing. https://martinfowler.com/eaaDev/EventSourcing.html, décembre 2005. Consulté en mars 2021.

[10] Chris Richardson. Pattern : Event sourcing. https://microservices.io/patterns/data/event-sourcing.html. Consulté en mars 2021.

[11] Microsoft Docs. What is cqrs pattern ? https://docs.microsoft.com/en-us/azure/architecture/patterns/cqrs, novembre 2020. Consulté en mars 2021.

[12] Chris Richardson. Pattern : Command query responsibility segregation (cqrs). https://microservices.io/patterns/data/cqrs.html. Consulté en mars 2021.

[13] Eric Thérond. Cqrs, larchitecture aux deux visages (partie 1). https://blog.octo.com/cqrs-larchitecture-aux-deux-visages-partie-1/, novembre 2011. Consulté en mars 2021.

[14] François Saulnier. Cqrs, larchitecture aux deux visages (partie 2). https://blog.octo.com/cqrs-larchitecture-aux-deux-visages-partie2/, juin 2012. Consulté en mars 2021.

[15] Martin Fowler. Cqrs. https://martinfowler.com/bliki/CQRS.html, juillet 2011. Consulté en mars 2021.

[16] Apache Software Foundation. Apache kafka. https://kafka.apache.org/. Consulté en février 2021.

[17] Apache Software Foundation. Design. https://kafka.apache.org/documentation.html#design. Consulté en mars 2021.

[18] Apache Software Foundation. Persistence. https://kafka.apache.org/documentation.html#persistence. Consulté en mars 2021.

[19] Loïc Divad. Getting started with scala and apache kafka. https://www.confluent.io/blog/kafka-scala-tutorial-for-beginners/, décembre 2020. Consulté en avril 2021.

[20] Apache Software Foundation. Main concepts and terminology. https://kafka.apache.org/documentation.html#intro_concepts_and_terms. Consulté en mars 2021.

[21] Apache Software Foundation. Kafka streams. https://kafka.apache.org/28/documentation/streams/. Consulté en mars 2021.

[22] Amazon. Amazon kinesis. https://aws.amazon.com/fr/kinesis/. Consulté en mars 2021.

[23] Google. Google pub/sub. https://cloud.google.com/pubsub. Consulté en mars 2021.

[24] Microsoft Azure. Service bus. https://azure.microsoft.com/en-us/services/service-bus/. Consulté en mars 2021.

[25] Apache Software Foundation. Activemq - flexible & powerful open source multi-protocol messaging. https://activemq.apache.org/. Consulté en mars 2021.

[26] Victor Alekseev. Some detailed differences between kafka and activemq. https://blog.devgenius.io/some-detailed-differences-between-kafka-and-activemq-d40163cb2ac4, mai 2020. Consulté en mars 2021.

[27] RabbitMQ. Messaging that just works - rabbitmq. https://www.rabbitmq.com/. Consulté en mars 2021.

[28] Lovisa johansson. When to use rabbitmq or apache kafka. https://www.cloudamqp.com/blog/when-to-use-rabbitmq-or-apache-kafka.html, décembre 2019. Consulté en mars 2021.

[29] Elastic. Suite elk : Elasticsearch, logstash et kibana. https://www.elastic.co/fr/what-is/elk-stack. Consulté en mars 2021.

[30] Elastic. Logstash : collecte, transformation et analyse de logs | elastic. https://www.elastic.co/fr/logstash. Consulté en mars 2021.

[31] Suyog Rao ; Tal Levy. Un soupçon de kafka pour la suite elastic : Partie 1. https://www.elastic.co/fr/blog/just-enough-kafka-for-the-elastic-stack-part1, mai 2016. Consulté en mars 2021.

[32] Kai Waehner. Apache kafka vs. enterprise service bus (esb) - friends, enemies or frenemies ? https://www.confluent.io/blog/apache-kafka-vs-enterprise-service-bus-esb-friends-enemies-or-frenemies/, juillet 2018. Consulté en mars 2021.

[33] Debezium. Debezium. https://debezium.io. Consulté en avril 2021.

[34] Debezium. Debezium architecture. https://debezium.io/documentation/reference/1.5/architecture.html, août 2020. Consulté en avril 2021.

[35] Oracle. The binary log. https://dev.mysql.com/doc/refman/8.0/en/binary-log.html. Consulté en avril 2021.

[36] Apache Software Foundation. Welcome to apache avro ! http://avro.apache.org/. Consulté en avril 2021.

[37] Confluent Inc. ksqldb : The database purpose-built for stream processing applications. https://ksqldb.io/. Consulté en mai 2021.

[38] Michael Noll. Streams and tables in apache kafka : A primer. https://www.confluent.io/blog/kafka-streams-tables-part-1-event-streaming/, janvier 2020. Consulté en mai 2021.

[39] Apache Software Foundation. Core concepts. https://kafka.apache.org/28/documentation/streams/core-concepts. Consulté en mai 2021.

[40] Confluent Inc. An overview of ksqldb. https://ksqldb.io/overview.html. Consulté en mai 2021.

[41] Apache Software Foundation. Apache spark is a unified analytics engine for large-scale data processing. https://spark.apache.org/. Consulté en mars 2021.

[42] Confluent Inc. Materialized cache. https://docs.ksqldb.io/en/latest/tutorials/materialized/, mars 2021. Consulté en mai 2021.

[43] Confluent Inc. Streaming etl pipeline. https://docs.ksqldb.io/en/latest/tutorials/etl/, mars 2021. Consulté en mai 2021.

[44] Confluent Inc. Event-driven microservice. https://docs.ksqldb.io/en/latest/tutorials/event-driven-microservice/, mars 2021. Consulté en mai 2021.

[45] Wikipédia. Association internationale du transport aérien. https://fr.wikipedia.org/wiki/Association_internationale_du_transport_a%C3%A9rien, avril 2021. Consulté en avril 2021.

[46] Mockaroo. Mockaroo - random data generator and api mocking tool | json / csv / sql / excel. https://www.mockaroo.com/. Consulté en février 2021.

[47] Docker. Overview of docker compose. https://docs.docker.com/compose/, avril 2021. Consulté en février 2021.

[48] Wikipédia. Java database connectivity. https://fr.wikipedia.org/wiki/Java_Database_Connectivity, novembre 2018. Consulté en mars 2021.

[49] Oracle. Using transactions. https://docs.oracle.com/javase/tutorial/jdbc/basics/transactions.html. Consulté en mars 2021.

[50] Eddie Carrasco. Mysql : When to order before group. https://eddies-shop.medium.com/mysql-when-to-order-before-group-13d54d6c4ebb, septembre 2020. Consulté en avril 2021.

[51] Oracle. Window function concepts and syntax. https://dev.mysql.com/doc/refman/8.0/en/window-functions-usage.html. Consulté en avril 2021.

[52] Ignacio L. Bisso. What is the mysql over clause ? https://learnsql.com/blog/over-clause-mysql/, septembre 2020. Consulté en avril 2021.

[53] MySQL Tutorial. Mysql ROW_NUMBER function. https://www.mysqltutorial.org/mysql-window-functions/mysql-row_number-function/. Consulté en avril 2021.

[54] Debezium. Tutorial. https://debezium.io/documentation/reference/1.5/tutorial.html, novembre 2020. Consulté en avril 2021.

[55] Debezium Authors. Debezium tutorial. https://github.com/debezium/debezium-examples/tree/master/tutorial, avril 2021. Consulté en avril 2021.

[56] Johannes Rudolph. spray-json : A lightweight, clean and simple json implementation in scala. https://github.com/spray/spray-json, novembre 2020. Consulté en mai 2021.

[57] Confluent Inc. How it works - ksqldb and kafka streams. https://docs.ksqldb.io/en/latest/operate-and-deploy/how-it-works/#ksqldb-and-kafka-streams. Consulté en mai 2021.

[58] Ben Stopford. Chain services with exactly-once guarantees. https://www.confluent.io/blog/chain-services-exactly-guarantees/, juillet 2017. Consulté en mai 2021.

[59] Michael Seifert. Building transactional systems using apache kafka. https://www.confluent.io/blog/transactional-systems-with-apache-kafka/, août 2019. Consulté en mai 2021.

[60] Apurva Mehta ; Jason Gustafson. Transactions in apache kafka. https://www.confluent.io/blog/transactions-apache-kafka/, novembre 2017. Consulté en mai 2021.

[61] Neil Avery. Journey to event driven part 1 : Why event-first programming changes everything. https://www.confluent.io/blog/journey-to-event-driven-part-1-why-event-first-thinking-changes-everything/, janvier 2019. Consulté en mai 2021.

[62] Kevin Petrie. Applying the acid test to apache kafka. https://www.eckerson.com/articles/applying-the-acid-test-to-apache-kafka, novembre 2018. Consulté en mai 2021.

[63] Kai Waehner. Can apache kafka replace a database ? the 2020 update. https://www.kai-waehner.de/blog/2020/03/12/can-apache-kafka-replace-database-acid-storage-transactions-sql-nosql-data-lake/, mars 2020. Consulté en mai 2021.