
DÉPARTEMENT D'INFORMATIQUE
B.U.T. INFORMATIQUE

PROJET DE BASES DE DONNÉES AVANCÉES

# Simulation d'une plateforme de livraison en temps réel avec Redis et MongoDB

**PRÉSENTÉ PAR :**

David Diema

**ENCADRANT :**

Christophe Cerin

ANNÉE UNIVERSITAIRE 2025–2026

# Table des matières

# 1 Introduction

## 1.1 Contexte

De nos jours, les plateformes de livraison de repas comme UberEats, AlloResto ou encore Just Eat font clairement partie de notre quotidien. La clé de leurs succès est leur capacité à pouvoir gérer un flux de données très important en peu de temps. Le fait de pouvoir assurer une communication presque instantanée et fiable entre les différents acteurs de ces plateformes est fondamental. Pour ce faire, il est primordial d'avoir une architecture capable d'assurer une transmission rapide des données et une bonne gestion des changements d'état fréquents. On se questionne de plus en plus sur les limites des bases de données traditionnelles. Ces dernières organisent les informations en les séparant dans différentes tables, ce qui nécessite ensuite de les relier entre elles pour obtenir des données complètes, une opération qui peut être lente et complexe. Les bases de données NoSQL proposent une approche alternative : elles regroupent directement toutes les informations liées dans un même emplacement, sous forme de documents ou d'objets complets. Cette méthode permet d'accéder aux données plus rapidement, sans avoir à reconstituer les informations à partir de multiples sources dispersées.

## 1.2 Problématique

Dans le cadre de notre application de livraison de repas, on doit gérer le suivi d'une commande qui passe par plusieurs états (création, acceptation par le restaurant, préparation, livraison). Le problème, c'est que trois acteurs différents doivent être au courant de ces changements en temps réel : le client qui attend sa commande, le livreur qui doit savoir quand partir, et le restaurant qui prépare le plat.

La question qu'on se pose, est comment garantir que tout le monde ait les mêmes informations en temps réel, sans passer par des API REST classiques ? On veut tester deux approches NoSQL événementielles : les Change Streams de MongoDB d'un côté, et le système Pub/Sub de Redis de l'autre. MongoDB permet d'observer directement les changements dans la base de données, tandis que Redis fonctionne plutôt comme un système de messagerie entre les différents services.

## 1.3 Objectifs

1. Mettre en place un premier prototype (POC) utilisant MongoDB où les changements d'état sont gérés par les Change Streams. Sa fonction sera de montrer comment chaque acteur s'abonne aux modifications de la base de données pour mettre à jour sa vision de l'état de commande.
2. Implémenter un second prototype (POC) équivalent au premier mais en utilisant cette fois-ci Redis. Les changements d'état seront gérés par le système Pub/Sub (Publication/Subscription).
3. Développer une fonctionnalité de reporting du chiffre d'affaires pour chaque base de données.
4. Comparer les deux prototypes au niveau du code et de la complexité, puis déterminer lequel est le plus adapté pour répondre à notre problématique. On évaluera

également quelle solution facilite le mieux l'ajout d'une fonctionnalité supplémentaire : donner la possibilité au client d'annuler sa commande dans les 30 secondes qui suivent, à condition qu'elle n'ait pas encore été acceptée par un livreur.

**Résultat principal :** Au final, le résultat le plus marquant de ce projet concerne la fonctionnalité de reporting du chiffre d'affaires. En l'implémentant pour les deux systèmes, on s'est rendu compte que MongoDB est clairement supérieur à Redis pour l'analyse de données : MongoDB effectue le calcul directement dans la base avec son Framework d'Agrégation(`$match`, `$group`, `$sum`), tandis que Redis oblige à récupérer toutes les données et à faire les calculs manuellement en Python. Cette différence fondamentale, ainsi que d'autres critères de comparaison, seront analysés en détail dans la section 4.2.

**Annonce du plan :** Ce rapport s'articule autour de cinq sections principales. À la section 2, nous présentons l'environnement technique mis en place (Docker, MongoDB, Redis) ainsi que le protocole d'interaction asynchrone et la modélisation des données par dénormalisation. La section 3 détaille les résultats obtenus avec les deux implémentations (MongoDB Change Streams et Redis Pub/Sub), ainsi que la fonctionnalité de reporting du chiffre d'affaires développée pour chaque système. À la section 4, nous analysons et comparons les deux approches en termes de fiabilité, de performance d'analyse de données, et nous discutons des limites identifiées. Enfin, la section 5 présente nos conclusions sur le choix technologique le plus adapté à notre problématique de synchronisation temps réel.

# 2 Matériels et Méthodes

## 2.1 Environnement Technique

Pour ce projet, j'ai choisi de travailler avec Docker et Docker Compose pour conteneuriser l'ensemble de l'environnement. Utiliser Docker m'a vraiment simplifié la vie au niveau de l'installation et de la configuration. Par exemple, je n'ai pas besoin de gérer manuellement le démarrage de mes deux services (Redis et MongoDB) : avec un simple `docker-compose up`, je lance automatiquement les deux services dans un réseau isolé, et quand j'ai fini, un `docker-compose down` arrête et nettoie tout proprement.

Au-delà de la simplicité d'utilisation, Docker était aussi le choix logique pour garantir que mon projet fonctionne de la même manière sur n'importe quelle machine, que ce soit la mienne ou celle du professeur qui va l'évaluer. Ça évite les problèmes de conflit liés aux versions.

## 2.2 Installation et Configuration de MongoDB

Au lieu d'installer MongoDB directement sur ma machine, j'ai préféré le définir comme un service dans le fichier `docker-compose.yml` en utilisant l'image officielle `mongo`.

Toujours dans le même fichier, j'ai mappé le port par défaut de MongoDB (27017) avec `ports: - "27017:27017"` pour pouvoir y accéder depuis ma machine locale.

J'ai ensuite ajouté `command: mongod -replSet rs0 -bind_ip_all` pour forcer MongoDB à démarrer en mode Replica Set (que j'ai appelé `rs0`). Sans ça, impossible d'utiliser les Change Streams.

1. D'abord, j'ai démarré le service avec `docker-compose up -d`

2. Ensuite, il faut initialiser le Replica Set manuellement. J'ai exécuté cette commande en bash :

```
docker exec -it mongodb mongosh --eval "rs.initiate({_id: 'rs0',
members: [{_id: 0, host: 'localhost:27017'}]})"
```

Mes scripts Python peuvent ensuite se connecter avec l'URI *mongodb ://localhost :27017/ ?replicaSet=rs0.*

## 2.3 Installation et Configuration de Redis

Pour Redis, même principe que MongoDB : j'ai tout défini dans le `docker-compose.yml` avec l'image officielle `redis`. Ça m'évite de devoir l'installer sur ma machine et de me retrouver avec des conflits de versions.

Ensuite j'ai simplement fait la même chose qu'avec MongoDB, j'ai mappé le port par défaut puis j'ai exécuté la commande `docker compose up -d`.

Après ça Redis est directement utilisable. Mes scripts Python peuvent se connecter avec `redis.Redis(host='localhost', port=6379)` pour gérer les commandes et la communication Pub/Sub.

## 2.4 Protocole d'Interaction

Pour simuler le déroulement d'une commande, j'ai mis en place un système où les trois acteurs ne se parlent jamais directement. À la place, ils communiquent de manière asynchrone : soit en observant les changements d'état dans la base de données avec MongoDB, soit en s'envoyant des messages via Redis.

Le flux que j'ai choisi suit cette logique : le client crée sa commande, le livreur la récupère et la transmet au restaurant, le restaurant la prépare et prévient le livreur, puis le livreur livre au client.

Ce protocole est visualisé par la figure 1 ci-dessous :

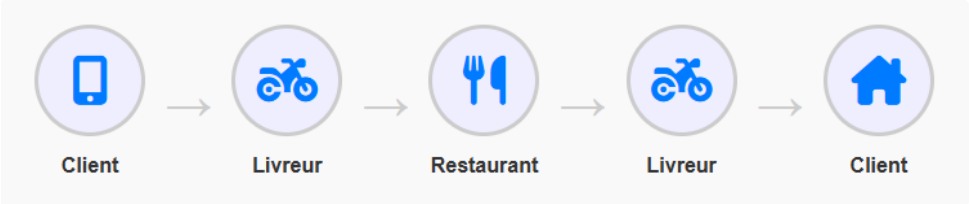


FIGURE 1 – Protocole d'interaction entre les trois acteurs (Client, Livreur, Restaurant)

Ensuite, ce protocole est détaillé par le diagramme de séquence de la figure 2 :

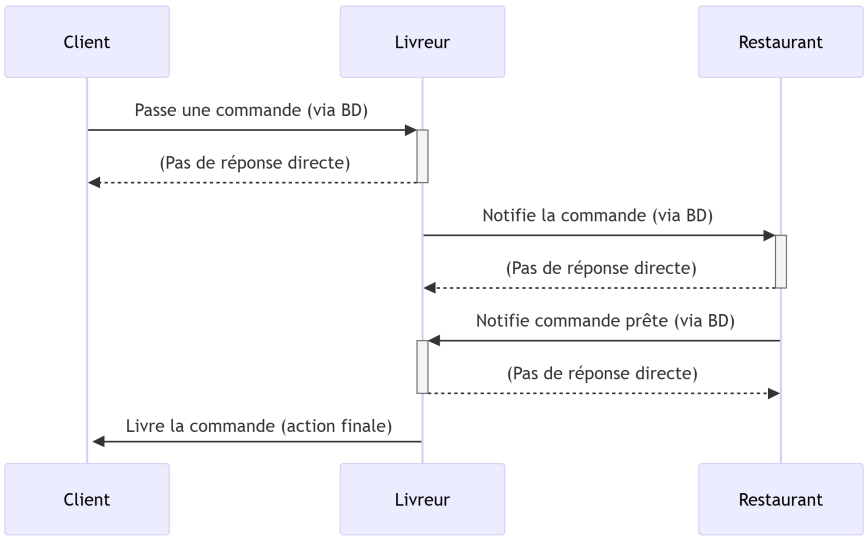


FIGURE 2 – Diagramme de séquence illustrant le protocole d'interaction asynchrone.

Ce diagramme représente le flux de communication entre les trois acteurs principaux de notre système. On observe que le client initie la commande, qui passe ensuite par le livreur pour être transmise au restaurant. Une fois préparée, la commande suit le chemin inverse : du restaurant vers le livreur, puis du livreur vers le client. À chaque étape, le statut de la commande est mis à jour (en_attente_livreur, en_preparation, prete_a_livrer, en_cours_de_livraison, livree). Ce protocole asynchrone permet aux acteurs de réagir aux changements sans communication directe entre eux.

## 2.5 Modélisation des données

Pour la structure des données, j'ai appliqué le principe de dénormalisation, qui est un peu la philosophie de base du NoSQL. L'idée, c'est de dupliquer volontairement certaines informations au lieu de les séparer dans plusieurs tables comme on ferait en SQL classique. Concrètement, ça veut dire que chaque service a toutes les infos dont il a besoin directement sous la main, sans avoir à aller chercher des données ailleurs. C'est beaucoup plus rapide pour la lecture, et ça évite les jointures compliquées qu'on aurait avec une base relationnelle traditionnelle.

### 2.5.1 Préparation des Données Initiales

Pour mettre en pratique cette approche et alimenter mes simulations, j'ai créé un script Python (`preparer_donnees.py`) qui traite le fichier CSV `all_restaurants_menu.csv` récupéré sur Kaggle. Le script charge d'abord le CSV avec `pandas`, puis nettoie les données en gardant seulement les colonnes utiles et en convertissant les prix dans le bon format. J'ai ensuite sélectionné 50 restaurants au hasard pour avoir un échantillon raisonnable. C'est à cette étape que j'applique concrètement la dénormalisation : je regroupe tous les plats de chaque restaurant ensemble, plutôt que de les garder séparés. Le script génère aussi 10 livreurs fictifs pour les besoins de la simulation. Tout est sauvegardé dans un fichier `denormalisation.json` qui me sert de base pour lancer les simulations.

Voici la structure d'un restaurant dans le fichier `denormalisation.json` :

```
{
"restaurants": [
    {
        "restaurant_id": "1353",
        "nom": "Restaurant 1353",
        "menu": [
            {
                "menu_item": "Carne Asada Burrito Bowl",
                "price_float": 20.0
            },
            {
                "menu_item": "Enchilada Plate",
                "price_float": 20.0
            },
            {
                "menu_item": "Bean Crisp",
                "price_float": 11.0
            },
            {
                "menu_item": "Twisted Cowgirl Dip",
                "price_float": 12.0
            },
            {
                "menu_item": "Grilled Chicken Burrito Bowl",
                "price_float": 18.0
            },
            {
                "menu_item": "Twisted Cowgirl Dip",
                "price_float": 12.0
            },
            {
                "menu_item": "Taquitos (6 pcs)",
                "price_float": 10.0
            },

```

FIGURE 3 – Structure d'un restaurant dans le fichier denormalisation.json.

Cette structure illustre concrètement le principe de dénormalisation appliqué dans ce projet. On peut voir que toutes les informations du restaurant (ID, nom, ville) et l'intégralité de son menu sont regroupées dans un seul objet JSON. Chaque plat contient directement son nom et son prix, évitant ainsi le besoin de faire des jointures ou des requêtes multiples pour obtenir ces informations. Cette approche permet au script client de sélectionner un restaurant et un plat en une seule opération de lecture.

### 2.5.2 Schéma des données MongoDB

J'utilise deux collections principales :

**Collection restaurants** (Peuplée par `preparer_donnees.py`, utilisée par `rapport_ventes.py`)

```
{
    "_id": ObjectId("..."),
    "restaurant_id": "string",
    "menu_item": "string",
    "price": "string" // Prix brut du CSV
}
```

**Collection commandes**

```
{
    "_id": ObjectId("..."),
    "client_id": "string",
    "restaurant_id": "string",
    "livreur_id": "string" | null,
    "plat": "string",
    "prix": float,
    "statut": "string",
    "timestamp": float
}
```

### 2.5.3 Schéma des données Redis

Avec Redis, j'utilise deux mécanismes différents :

**Pour le stockage (Hashes)** Chaque commande est stockée avec une clé du type `commande:{uuid}` et contient ces infos :

```
{
    "command_id": "string",
    "client_id": "string",
    "restaurant_id": "string",
    "livreur_id": "string",
    "plat": "string",
    "prix": "string",
    "statut": "string"
}
```

**Pour la communication (Pub/Sub)** J'ai créé plusieurs canaux de messagerie :
- `nouvelles_commandes` : pour diffuser les nouvelles commandes à tous les livreurs
- `restaurant:{id}` : pour qu'un restaurant spécifique reçoive ses commandes
- `livreur:{id}` : pour qu'un livreur reçoive ses notifications personnelles

# 3 Résultats

## 3.1 Simulation avec MongoDB & Change Streams

Pour cette première implémentation, j'utilise les Change Streams de MongoDB pour notifier les acteurs dès qu'un changement pertinent survient dans la collection `commandes`.

Concrètement, mes scripts `livreur.py` et `restaurant.py` ouvrent chacun un curseur avec `watch()` et un pipeline `$match` spécifique pour ne réagir qu'aux événements qui les concernent. J'ai découvert que l'option `full_document='updateLookup'` était indispensable pour récupérer l'état complet du document après chaque mise à jour.

Voici comment j'ai configuré le pipeline de surveillance dans `restaurant.py` :

```
pipeline = [
    {'$match': {
        'operationType': 'update',
        'fullDocument.statut': 'en_preparation'
    }}
]
try:
    with commandes_collection.watch(pipeline,
                                    full_document='updateLookup'
    ↪ ) as stream:
        # ...
```

Quand je démarre les scripts (`restaurant.py`, un ou plusieurs `livreur.py`, puis `client.py`), le client commence par insérer une commande avec le statut `en_attente_livreur`. Un des livreurs actifs détecte l'insertion via son Change Stream et tente de s'assigner la course avec une opération `update_one` atomique (qui vérifie que le champ `livreur_id` est bien `None`). Le livreur qui réussit à prendre la commande met à jour le statut à `en_preparation`.

Le script `restaurant.py`, qui écoute tous les changements, détecte cette mise à jour, vérifie si l'ID correspond à un de ses restaurants, simule un temps de préparation et passe le statut à `prete_a_livrer`. Le livreur assigné capte ce changement via son Change Stream, met le statut à `en_cours_de_livraison`, simule le trajet, et finalise avec le statut `livree`.

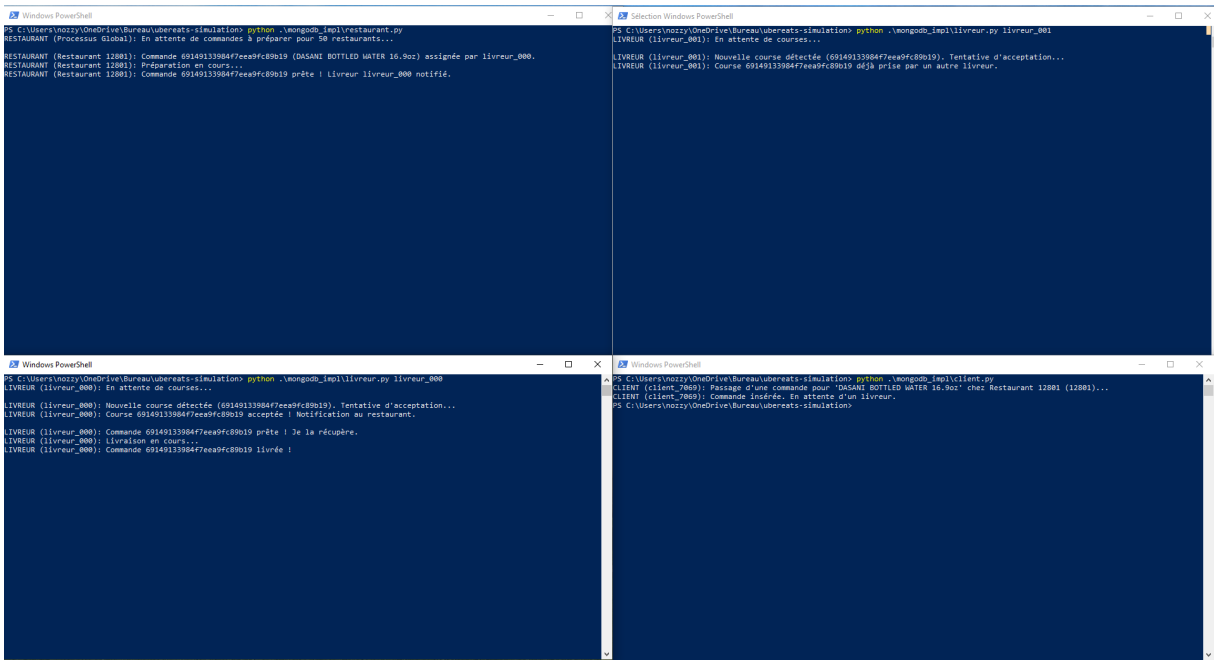


FIGURE 4 – Captures d'écran illustrant le déroulement de la simulation MongoDB.

Ces captures montrent l'exécution réelle de la simulation avec MongoDB. On observe : le terminal du client en bas à droite, avec l'insertion de la commande. Les terminaux des livreurs en haut à droite et en bas à gauche, avec la détection et l'acceptation via Change Stream. Le terminal du restaurant en haut à gauche, avec la simulation de préparation, et le retour au terminal du livreur pour la livraison. Chaque acteur réagit en temps réel aux changements d'état grâce aux Change Streams, sans jamais communiquer directement entre eux.

## 3.2 Simulation avec Redis & Pub/Sub

Pour Redis, j'ai pris une approche différente : j'utilise le système Pub/Sub pour les notifications et les Hashes pour stocker l'état de chaque commande. Le client publie l'ID de la nouvelle commande sur le canal `nouvelles_commandes`. Les livreurs s'abonnent à ce canal général plus leur canal privé `livreur:{id}`. Le restaurant s'abonne aux canaux de tous les restaurants qu'il gère (`restaurant:{id}`).

Voici comment ça se passe dans `livreur.py` :

```
pubsub = r.pubsub()
pubsub.subscribe('nouvelles_commandes')
pubsub.subscribe(f"livreur:{livreur_id}") # Son canal prive
print(f"LIVREUR ({livreur_id}): En attente de courses...")

for message in pubsub.listen():
    if message['type'] != 'message':
        continue
    # ...
```

Le client crée le Hash de la commande et publie son ID sur `nouvelles_commandes`. Un livreur reçoit le message, vérifie le Hash (surtout que le champ `livreur_id` est bien à `"None"`), et tente de s'assigner la course en mettant à jour le Hash. Pour cette démo, j'ai simplifié en ne gérant pas l'atomicité comme avec MongoDB.

Le livreur qui récupère la course met le statut à `en_preparation` et publie l'ID sur le canal `restaurant:{id}` correspondant. Le script `restaurant.py` reçoit le message sur son canal, vérifie le Hash, simule la préparation, passe le statut à `prete_a_livrer` et publie sur le canal privé du livreur (`livreur:{id}`).

Le livreur concerné capte le message, met le statut à `en_cours_de_livraison`, simule la livraison et finalise avec `livree`.

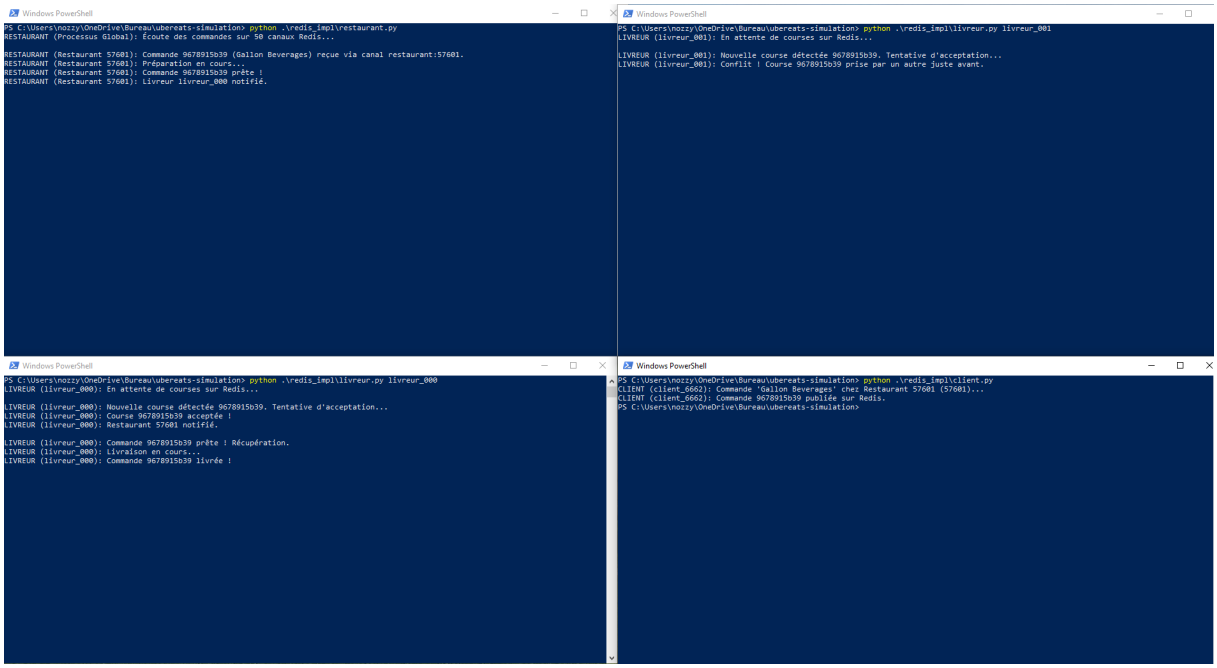


FIGURE 5 – Captures d'écran illustrant le déroulement de la simulation Redis.

Ces captures présentent l'exécution de la simulation avec Redis Pub/Sub. Comme pour MangoDB, on observe les terminaux du client, du livreur et du restaurant communiquant via les différents canaux de messagerie. Le système Pub/Sub assure une communication rapide et événementielle entre les acteurs via les canaux nouvelles_commandes, restaurant :{id} et livreur :{id}.

## 3.3 Reporting de Ventes

### 3.3.1 Reporting avec MongoDB (Agrégation Native)

Pour MongoDB, j'ai créé le script `rapport_ventes.py` qui se connecte à la collection `commandes` et utilise le Framework d'Agrégation intégré à MongoDB. L'avantage ici, c'est que tout le calcul se fait directement dans la base de données.

```
pipeline = [
    {
        '$match': { 'statut': 'livree' }
    },
    {
        '$group': {
            '_id': None,
            'chiffre_affaires_total': { '$sum': '$prix' },
            'nombre_commandes': { '$sum': 1 }
        }
    }
]
```

Cette approche est vraiment efficace puisque le serveur MongoDB gère l'intégralité du traitement. C'est optimisé pour traiter de gros volumes de données sans avoir à transférer toutes les informations vers l'application Python.

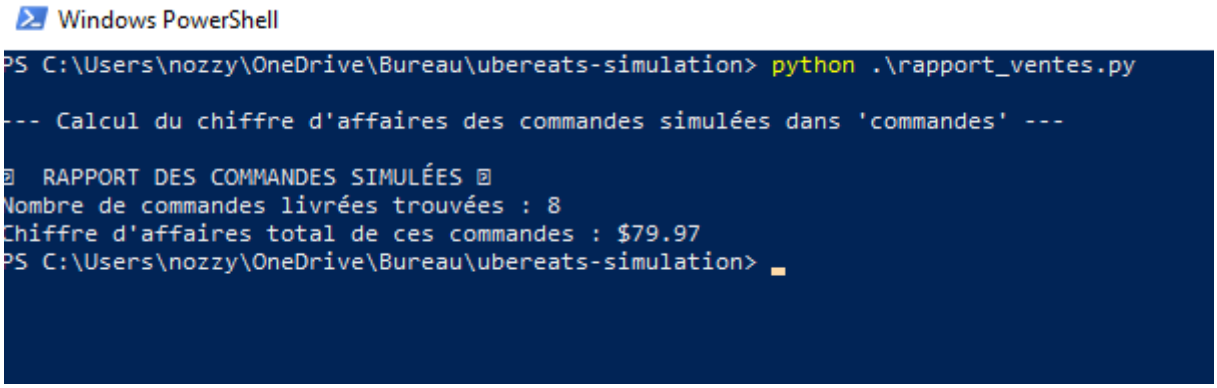


FIGURE 6 – Résultat du calcul du chiffre d'affaires avec MongoDB.

Cette capture montre le résultat de l'exécution du script `rapport_ventes.py`. Le Framework d'Agrégation de MongoDB a calculé automatiquement le chiffre d'affaires total ainsi que le nombre de commandes livrées. On peut voir que toutes les commandes ayant le statut "livree" ont été prises en compte dans le calcul. L'opération s'est effectuée entièrement côté serveur MongoDB, démontrant l'efficacité de cette approche pour l'analyse de données.

### 3.3.2 Reporting avec Redis (Calcul Côté Application)

Comme Redis n'a pas de système d'agrégation complexe comme MongoDB, j'ai dû faire le calcul dans mon script Python (`rapport_ventes_redis.py`).

Le processus est plus manuel : le script commence par récupérer toutes les clés des commandes (avec `KEYS commande:*`), puis il boucle sur chacune d'elles. Pour chaque clé, il récupère le Hash complet avec `HGETALL`, vérifie si le statut est `"livree"`, et si c'est le cas, il convertit le prix (qui est stocké en string dans Redis) en `float` et l'ajoute à un total qui tourne en Python.

Ça marche très bien, mais c'est clairement moins performant que l'approche MongoDB. Le problème, c'est qu'il faut rapatrier toutes les données de Redis vers le script pour faire les calculs, alors qu'avec MongoDB tout reste dans la base.

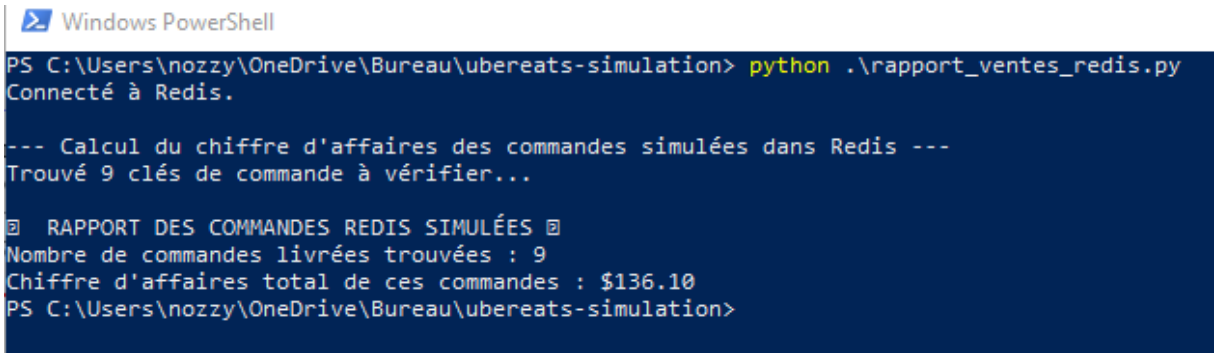


FIGURE 7 – Résultat du calcul du chiffre d'affaires avec Redis.

Cette capture présente le résultat du script rapport_ventes_redis.py. Contrairement à MongoDB, le calcul a dû être effectué côté application Python. Le script a parcouru toutes les clés de commandes, vérifié leur statut, et additionné les prix manuellement. Bien que le résultat soit identique à celui de MongoDB, on observe que cette approche nécessite plus de traitement côté client et transfère davantage de données sur le réseau.

# 4 Discussion

## 4.1 Interprétation des Résultats

### 4.1.1 Validation des Mécanismes Événementiels

Avec MongoDB les Change Streams ont bien fonctionné. Mon script `restaurant.py` qui écoute tous les changements et mes scripts `livreur.py` qui écoutent uniquement leurs propres commandes arrivent sans problème à capter les changements d'état (`en_preparation`, `prete_a_livrer`). Ça prouve que c'est un mécanisme fiable pour observer ce qui se passe dans la base. Le système réagit automatiquement dès qu'une donnée change.

Pareil pour la simulation Redis, le Pub/Sub est vraiment rapide. Les messages arrivent instantanément sur les bons canaux (`nouvelles_commandes`, `restaurant:{id}`), donc les acteurs peuvent réagir directement.

### 4.1.2 Validation du Modèle de Données Dénormalisé

L'utilisation du fichier `denormalisation.json` valide aussi ma stratégie. J'ai regroupé le menu de chaque restaurant directement dans l'objet restaurant, du coup mon script `client.py` peut choisir un restaurant et un plat en une seule lecture. Avec du SQL classique, j'aurais dû faire deux requêtes (une pour le restaurant, une pour ses plats). C'est donc un gain de performance direct.

## 4.2 Analyse Comparative

### 4.2.1 Analyse du Reporting (Chiffre d'Affaires)

L'implémentation des scripts de reporting (section 3.3) a mis en lumière une différence majeure entre les deux systèmes :

MongoDB est clairement conçu pour l'analyse de données. Son Framework d'Agrégation (`$match`, `$group`, `$sum`) est un outil natif, puissant et optimisé. Le calcul du chiffre

d'affaires est délégué directement au serveur de base de données, qui peut traiter des millions de commandes de manière efficace.

Redis, par contre, n'a pas d'outil d'agrégation intégré. Pour calculer le chiffre d'affaires (`rapport_ventes_redis.py`), j'ai dû écrire un script Python qui :

— Récupère toutes les clés de commande (`KEYS commande:*`)
— Pour chaque clé, fait une nouvelle requête pour récupérer le Hash (`HGETALL`)
— Effectue le calcul (filtrage, addition) côté application

Cette méthode devient vraiment inefficace à grande échelle. Elle surcharge l'application et nécessite de transférer d'énormes quantités de données sur le réseau.

On peut donc en conclure que pour l'analyse de données, MongoDB est clairement supérieur.

### 4.2.2 Analyse de Fiabilité

Pour évaluer la fiabilité des deux systèmes, j'ai analysé le cas d'une annulation de commande par le client. Le scénario est simple : le client peut annuler dans les 30 secondes suivant sa commande, mais seulement si aucun livreur n'a encore accepté la course.

La vérification des 30 secondes est une logique applicative assez simple (juste une comparaison de timestamps) qui serait gérée de la même manière dans les deux systèmes. Le vrai défi technique, c'est la seconde condition : comment être sûr que la commande n'est pas acceptée par un livreur exactement au même moment où le client l'annule ? C'est ce qu'on appelle une "race condition" classique.

**Avec MongoDB** Le problème se résout assez élégamment grâce aux transactions ACID. Mon script livreur utilise déjà une opération `update_one` atomique pour s'assurer d'être le seul à prendre la course (il vérifie que `livreur_id` est `None`). Pareil pour l'annulation : après avoir vérifié les 30 secondes, le client utiliserait une opération atomique pour passer le statut à `"annulee"` uniquement si le statut est toujours `"en_attente_livreur"`.

MongoDB agit donc comme un arbitre fiable. La cohérence des données est garantie par la base elle-même. Mon code reste simple : je tente l'action et je vérifie si elle a réussi.

**Avec Redis** Là ça devient plus compliqué. Le Pub/Sub envoie juste des messages, il ne garantit rien sur l'état réel des données.

Mon script `livreur.py` actuel qui fait un `HGET` puis un `HSET` n'est pas atomique, donc il y a un vrai risque de conflit. Pour le rendre fiable, il faudrait que j'implémente manuellement une transaction optimiste avec `WATCH`, `MULTI`, et `EXEC`. Concrètement, le script doit "surveiller" le Hash de la commande avec `WATCH`, vérifier son statut (et le client ferait pareil pour annuler), puis lancer une transaction avec `MULTI`. Si quelqu'un d'autre modifie le Hash entre temps, ma transaction échoue et je dois recommencer.

Donc toute la gestion de la cohérence repose sur moi en tant que développeur. Le code devient nettement plus complexe et plus difficile à déboguer.

**Tableau Comparatif**

| Critère | MongoDB + Change Streams | Redis + Pub/Sub |
|---|---|---|
| **Mécanisme de notification** | Change Streams : observation des modifications dans la base de données | Pub/Sub : messagerie par canaux avec publication/abonnement |
| **Performance de synchronisation** | Temps réel, légèrement plus lent que Redis (quelques ms) | Très rapide, quasi-instantané (< 1 ms) |
| **Analyse de données** | Excellent : Framework d'Agrégation natif (`$match`, `$group`, `$sum`) | Limité : nécessite de rapatrier toutes les données côté application |
| **Gestion de la cohérence** | Transactions ACID atomiques natives | Transactions optimistes manuelles (`WATCH`/`MULTI`/`EXEC`) requises |
| **Complexité du code** | Simple : opérations atomiques gérées par la base | Complexe : logique de cohérence à implémenter manuellement |
| **Fiabilité des messages** | Garantie : les Change Streams rejouent les événements manqués | Aucune garantie : "fire-and-forget", messages perdus si destinataire indisponible |
| **Persistance des données** | Native : stockage permanent sur disque | En mémoire par défaut (nécessite Redis Streams pour persistance) |
| **Gestion des race conditions** | Automatique via transactions ACID | Manuelle, nécessite implémentation spécifique |
| **Scalabilité** | Replica Sets et Sharding natifs | Redis Cluster (mais Pub/Sub reste "fire-and-forget") |
| **Cas d'usage idéal** | Applications nécessitant cohérence forte + analyse de données | Applications de messagerie pure nécessitant vitesse maximale |
| **Adapté pour notre projet** | **Oui** - Couvre tous les besoins | **Partiellement** - Excellent pour la messagerie, limité pour l'analyse |

FIGURE 8 – Tableau de synthèse de l'analyse comparative MongoDB vs Redis

## 4.3 Limites et Recommandations

**Fiabilité de Redis Pub/Sub** Le plus gros problème de mon implémentation Redis, c'est le Pub/Sub lui-même. C'est du "fire-and-forget" : si mon script `restaurant.py` est planté ou en train de redémarrer quand un livreur publie un message, le message est perdu pour toujours. Du coup la commande ne serait jamais préparée, ça crée une incohérence grave.

On pourrait penser que la solution serait d'utiliser un Redis Cluster mais ça ne résout pas ce problème. Le Cluster c'est surtout pour la scalabilité (stocker plus de données qu'une seule machine peut gérer) et la haute disponibilité (continuer à fonctionner si un serveur tombe). Mais le Pub/Sub reste "fire-and-forget", c'est-à-dire qu'une fois le message envoyé, Redis ne garantit pas qu'il a été reçu et ne le conserve pas pour une livraison ultérieure..

La vraie solution ce serait d'utiliser Redis Streams. Un Stream c'est un journal de messages persistant. Le livreur utiliserait : :`XADD` au lieu de `PUBLISH` pour ajouter sa notification. Le restaurant lirait le Stream tranquillement. S'il plante et redémarre, il peut

juste demander à Redis tous les messages qu'il a loupés. Comme ça aucune commande n'est perdue.

**Gestion de la Concurrence Livreur (Redis)** Mon script `livreur.py` pour Redis avec son `HGET` puis `HSET` n'est pas atomique. En production, il faudrait vraiment utiliser les transactions optimistes (`WATCH`/`MULTI`/`EXEC`) pour être sûr qu'un seul livreur prenne la course.

**Scalabilité du Script Restaurant** Mon script `restaurant.py` (pour les deux bases) charge tous les restaurants en mémoire et écoute tout. Avec 50 000 restaurants, ça marcherait pas. En vrai, on utiliserait des microservices où chaque instance de script gère juste un sous-ensemble de restaurants.

# 5 Conclusion

Ce projet avait pour objectif de comparer deux approches NoSQL événementielles (MongoDB Change Streams et Redis Pub/Sub) pour synchroniser en temps réel l'état d'une commande entre des acteurs distribués, sans utiliser d'API REST classique.

Les deux prototypes développés ont démontré que les mécanismes événementiels fonctionnent efficacement pour la synchronisation temps réel. MongoDB avec ses Change Streams offre un "miroir réactif" de la base de données, tandis que Redis Pub/Sub agit comme un système de messagerie ultra-rapide.

Cependant, notre analyse comparative a révélé des différences fondamentales. **Pour l'analyse de données**, MongoDB est clairement supérieur grâce à son Framework d'Agrégation qui traite les données directement dans la base, alors que Redis nécessite de tout rapatrier côté application. **Pour la cohérence des données**, MongoDB garantit l'intégrité grâce aux transactions ACID atomiques, tandis que Redis demande au développeur d'implémenter manuellement des transactions optimistes (`WATCH`/`MULTI`/`EXEC`), augmentant considérablement la complexité du code.

Concernant la fonctionnalité d'annulation de commande dans les 30 secondes, MongoDB s'adapte naturellement grâce à ses opérations atomiques, alors que Redis nécessiterait une refonte complète de la logique de concurrence.

**En conclusion, pour notre problématique de plateforme de livraison nécessitant à la fois de la synchronisation temps réel ET de l'analyse de données, MongoDB s'impose comme la solution la plus adaptée.** Redis reste une excellente option pour des cas d'usage purement orientés messagerie où la persistance et l'analyse de données ne sont pas critiques.

Une amélioration future du projet Redis serait de remplacer Pub/Sub par Redis Streams pour garantir la livraison des messages, résolvant ainsi la principale limite identifiée.

# 6 Références

1. Dataset Kaggle - Restaurant Menus : https://www.kaggle.com/datasets/
2. Documentation MongoDB - Change Streams : https://www.mongodb.com/docs/manual/changeStreams/

3. Documentation Redis - Pub/Sub : https://redis.io/docs/latest/develop/pubsub/
4. Documentation Redis - Transactions : https://redis.io/docs/latest/develop/using-commands/transactions/
5. Documentation Docker : https://docs.docker.com/
6. Documentation Redis Streams : https://redis.io/docs/latest/develop/data-types/streams/

# 7 Annexes

## 7.1 Annexe A : Fichier docker-compose.yml

```yaml
version: '3.8'
services:
  mongodb:
    image: mongo
    container_name: mongodb
    ports:
      - "27017:27017"
    command: mongod --replSet rs0 --bind_ip_all

  redis:
    image: redis
    container_name: redis
    ports:
      - "6379:6379"
```

## 7.2 Annexe B : Structure du fichier denormalisation.json (extrait)


```json
{
  "restaurants": [
    {
      "restaurant_id": "rest_001",
      "name": "Le Petit Bistro",
      "city": "Paris",
      "menu": [
        {
          "menu_item": "Burger Classic",
          "price": "12.50"
        },
        {
          "menu_item": "Salade Caesar",
          "price": "9.90"
        }
      ]
    }
  ],
  "livreurs": [
    {
      "livreur_id": "livreur_001",
      "nom": "Jean Dupont"
    }
  ]
}
```

## 7.3 Annexe C : Code du pipeline d'agrégation MongoDB complet

```python
from pymongo import MongoClient

# Connexion a MongoDB
client = MongoClient('mongodb://localhost:27017/?replicaSet=rs0')
db = client['livraison']
commandes_collection = db['commandes']

# Pipeline d'agregation
pipeline = [
    {
        '$match': { 'statut': 'livree' }
    },
    {
        '$group': {
            '_id': None,
            'chiffre_affaires_total': { '$sum': '$prix' },
            'nombre_commandes': { '$sum': 1 }
        }
    }
]

# Execution
resultat = list(commandes_collection.aggregate(pipeline))

if resultat:
    print(f"Chiffre d'affaires total: {resultat[0]['
    ↪ chiffre_affaires_total']} EUR")
    print(f"Nombre de commandes livrees: {resultat[0]['
    ↪ nombre_commandes']}")
else:
    print("Aucune commande livree trouvee.")
```

## 7.4 Annexe D : Extrait du code Change Stream MongoDB

```python
from pymongo import MongoClient
import time

client = MongoClient('mongodb://localhost:27017/?replicaSet=rs0')
db = client['livraison']
commandes_collection = db['commandes']

# Pipeline pour filtrer les evenements
pipeline = [
    {'$match': {
        'operationType': 'update',
        'fullDocument.statut': 'en_preparation'
    }}
]

print("Restaurant en attente de commandes...")

try:
    with commandes_collection.watch(pipeline,
                                    full_document='updateLookup'
    ) as stream:
        for change in stream:
            commande = change['fullDocument']
            print(f"Nouvelle commande a preparer: {commande['plat
    '']}")

            # Simulation de la preparation
            time.sleep(3)

            # Mise a jour du statut
            commandes_collection.update_one(
                {'_id': commande['_id']},
                {'$set': {'statut': 'prete_a_livrer'}}
            )
            print(f"Commande prete: {commande['plat']}")
except KeyboardInterrupt:
    print("\nArret du restaurant.")
```

## 7.5 Annexe E : Extrait du code Pub/Sub Redis

```python
import redis
import json
import time

r = redis.Redis(host='localhost', port=6379, decode_responses=
    ↪ True)

# ID du livreur
livreur_id = "livreur_001"

# Abonnement aux canaux
pubsub = r.pubsub()
pubsub.subscribe('nouvelles_commandes')
pubsub.subscribe(f"livreur:{livreur_id}")

print(f"LIVREUR ({livreur_id}): En attente de courses...")

for message in pubsub.listen():
    if message['type'] != 'message':
        continue

    channel = message['channel']
    commande_id = message['data']

    if channel == 'nouvelles_commandes':
        # Recuperer les details de la commande
        commande = r.hgetall(f"commande:{commande_id}")

        if commande.get('livreur_id') == "None":
            # Tenter de prendre la commande
            r.hset(f"commande:{commande_id}",
                   'livreur_id', livreur_id)
            r.hset(f"commande:{commande_id}",
                   'statut', 'en_preparation')

            # Publier au restaurant
            restaurant_id = commande.get('restaurant_id')
            r.publish(f"restaurant:{restaurant_id}", commande_id)

            print(f"Course acceptee: {commande.get('plat')}")

    elif channel == f"livreur:{livreur_id}":
        # Commande prete a livrer
        time.sleep(2)  # Simulation du trajet

        r.hset(f"commande:{commande_id}",
               'statut', 'livree')
        print(f"Commande livree: {commande_id}")
```

## 7.6 Annexe F : Lien vers le projet

Le code source complet de ce projet (incluant les scripts Python, le fichier docker-compose.yml et les données de test ) est disponible sur Google Drive à l'adresse suivante :

https://drive.google.com/file/d/13w1peGC-770jKbMf19NK6op7nXc5F74I/view?usp=sharing

Le dossier contient l'ensemble des fichiers nécessaires pour reproduire les simulations présentées dans ce rapport.

## 7.7 Annexe G : Guide de Démarrage et Commandes d'Exécution

Ce guide décrit les étapes et les commandes nécessaires pour installer l'environnement et exécuter les simulations après avoir téléchargé les fichiers du projet.

### 7.7.1 1. Prérequis

— Docker Desktop (en cours d'exécution)
— Python 3.x et pip
— Le fichier `all_restaurants_menu.csv` (de Kaggle) placé à la racine du projet.

### 7.7.2 2. Installation et Lancement des Services

Ces commandes installent les dépendances Python et lancent les conteneurs Docker pour MongoDB et Redis.

```
# 1. Installer les d pendances Python (pymongo, redis, pandas)
pip install -r requirements.txt

# 2. Lancer les conteneurs Docker (MongoDB et Redis)
docker-compose up -d

# 3. (IMPORTANT) Initialiser le Replica Set MongoDB (n cessaire
   ↪ pour les Change Streams)
docker exec -it mongodb mongosh --eval "rs.initiate({_id: 'rs0',
   ↪ members: [{_id: 0, host: 'localhost:27017'}]})"
```

### 7.7.3 3. Préparation des Données

Cette commande exécute le script qui lit le CSV de Kaggle et génère le fichier denormalisation.json utilisé par les simulations.

```
# 4. Ex cuter le script de pr paration des donn es
python preparer_donnees.py
```

### 7.7.4 4. Exécution de la Simulation MongoDB

Il est nécessaire d'ouvrir plusieurs terminaux.

```
# (Optionnel) Vider la collection des simulations precedentes
docker exec -it mongodb mongosh --eval "db.getSiblingDB('
    ↪ uber_eats').commandes.drop()"

# Terminal A : Lancer le script restaurant ( coute  globale)
python mongodb_impl/restaurant.py

# Terminal B : Lancer un livreur
python mongodb_impl/livreur.py livreur_000

# Terminal C (Optionnel) : Lancer un deuxieme livreur
python mongodb_impl/livreur.py livreur_001

# Terminal D : Lancer un client (cree une commande aleatoire)
python mongodb_impl/client.py
```

### 7.7.5 5. Exécution de la Simulation Redis

La procédure est similaire, dans des terminaux séparés.

```
# (Optionnel) Vider la base de donnees Redis
docker exec -it redis redis-cli FLUSHALL

# Terminal A : Lancer le script restaurant (ecoute globale)
python redis_impl/restaurant.py

# Terminal B : Lancer un livreur
python redis_impl/livreur.py livreur_000

# Terminal C : Lancer un client (cree une commande aleatoire)
python redis_impl/client.py
```

### 7.7.6 6. Lancement des Scripts de Reporting

Ces scripts peuvent être lancés après avoir exécuté les simulations (au moins une fois) pour générer des données.

```
# Pour calculer le chiffre d'affaires des commandes MongoDB
    ↪ livrees
python rapport_ventes.py

# Pour calculer le chiffre d'affaires des commandes Redis livrees
python rapport_ventes_redis.py
```