TP 03
Bases des données orientées clé-valeur
Études de cas

Étude de cas 1 : Système de recommandation en temps réel

Imaginons une plateforme de commerce électronique qui souhaite mettre en place un système de recommandation en temps réel pour ses utilisateurs. Redis peut jouer un rôle crucial dans cette implémentation.

Contexte

- Millions d'utilisateurs actifs
- Catalogue de produits en constante évolution
- Nécessité de recommandations personnalisées et rapides

Solution avec Redis

➢ Stockage des profils utilisateurs
  - Utilisation de hashes Redis pour stocker les informations utilisateur
  - Clé : user:{user_id}
  - Champs : préférences, historique d'achats, catégories favorites

➢ Suivi des produits consultés récemment
  - Utilisation de listes Redis pour chaque utilisateur
  - Clé : recent_views:{user_id}
  - Ajout des produits consultés avec LPUSH
  - Limitation de la taille avec LTRIM

➢ Calcul des recommandations
  - Utilisation de sets triés (sorted sets) pour stocker les scores de recommandation
  - Clé : recommendations:{user_id}

- Mise à jour des scores avec ZADD
- Récupération des meilleures recommandations avec ZREVRANGE

➢ Cache des informations produits

- Utilisation de hashes Redis pour stocker les détails des produits
- Clé : product:{product_id}
- Mise en cache pour un accès rapide

Questions d'exercice

1. **Question 1 : Mettre à jour le profil utilisateur**

   Écrivez une fonction nommée update_user_profile qui prend en entrée un identifiant utilisateur (user_id) et un dictionnaire de préférences (preferences). Cette fonction doit stocker les préférences de l'utilisateur dans une hash Redis avec la clé sous la forme user:{user_id}. Utilisez la commande Redis appropriée pour cette opération.

2. **Question 2 : Ajouter un produit consulté récemment**

   Écrivez une fonction nommée add_recent_view qui prend en entrée un identifiant utilisateur (user_id) et un identifiant de produit (product_id). Cette fonction doit ajouter le produit à la liste des produits récemment consultés par l'utilisateur dans Redis, en utilisant la clé recent_views:{user_id}. Limitez la liste aux 20 derniers produits consultés.

3. **Question 3 : Mettre à jour les recommandations**

   Écrivez une fonction nommée update_recommendations qui prend en entrée un identifiant utilisateur (user_id), un identifiant de produit (product_id) et un score (score). Cette fonction doit mettre à jour le score de recommandation pour le produit dans un set trié Redis avec la clé recommendations:{user_id}.

4. **Question 4 : Récupérer les meilleures recommandations**

Écrivez une fonction nommée get_top_recommendations qui prend en entrée un identifiant utilisateur (user_id) et un nombre (count, par défaut 5). Cette fonction doit retourner les meilleures recommandations (produits) pour l'utilisateur en utilisant le set trié recommendations:{user_id}.

5. **Question 5 : Mettre en cache les informations d'un produit**

   Écrivez une fonction nommée cache_product_info qui prend en entrée un identifiant de produit (product_id) et un dictionnaire contenant les informations du produit (product_info). Cette fonction doit stocker les informations du produit dans une hash Redis avec la clé product:{product_id}.

6. **Question 6 : Récupérer les informations d'un produit**

   Écrivez une fonction nommée get_product_info qui prend en entrée un identifiant de produit (product_id). Cette fonction doit retourner toutes les informations stockées dans la hash Redis avec la clé product:{product_id}.

Étude de cas 2 : Système de gestion de sessions pour une application web à grande échelle

Considérons maintenant une application web à forte charge qui nécessite une gestion efficace des sessions utilisateur.

Contexte

- Millions d'utilisateurs connectés simultanément
- Nécessité d'une haute disponibilité et d'une faible latence
- Besoin de scalabilité horizontale

Solution avec Redis

- ➢ Stockage des sessions
  - Utilisation de hashes Redis pour stocker les données de session
  - Clé : session:{session_id}
  - Champs : user_id, timestamp, données spécifiques à l'application
- ➢ Gestion de l'expiration
  - Utilisation de la commande EXPIRE pour définir une durée de vie sur les sessions
  - Renouvellement automatique de l'expiration à chaque activité de l'utilisateur
- ➢ Distribution de charge
  - Utilisation de Redis Cluster pour répartir les sessions sur plusieurs nœuds
  - Sharding basé sur la clé de session pour une distribution uniforme
- ➢ Persistance des données
  - Configuration de la persistance Redis pour éviter la perte de données en cas de redémarrage

Questions:

1. **Question 1 : Créer une session utilisateur**

   Écrivez une fonction nommée create_session qui prend en entrée un identifiant utilisateur (user_id) et des données supplémentaires (data). Cette fonction doit générer un identifiant de session unique, stocker les données de la session dans une hash Redis avec la clé sous la forme session:{session_id}, et définir une durée d'expiration pour la session de 1 heure. La fonction doit retourner l'identifiant de session créé.

2. **Question 2 : Récupérer les données de session**

   Écrivez une fonction nommée get_session qui prend en entrée un identifiant de session (session_id). Cette fonction doit récupérer les données de la session correspondante depuis Redis. Si la session existe, elle doit renvoyer les données sous forme de dictionnaire, tout

en renouvelant la durée d'expiration de la session. Si la session n'existe pas, la fonction doit retourner None.

3. **Question 3 : Mettre à jour les données de session**

   Écrivez une fonction nommée update_session qui prend en entrée un identifiant de session (session_id) et des données supplémentaires (data). Cette fonction doit mettre à jour les données de la session dans Redis et renouveler la durée d'expiration de la session.

4. **Question 4 : Supprimer une session**

   Écrivez une fonction nommée delete_session qui prend en entrée un identifiant de session (session_id). Cette fonction doit supprimer la session correspondante de Redis

Étude de cas 3 : Système de limitation de débit (Rate Limiting) pour une API

Examinons maintenant comment Redis peut être utilisé pour implémenter un système de limitation de débit pour une API.

Contexte

- API publique avec des limites d'utilisation par utilisateur
- Nécessité de contrôler le nombre de requêtes par minute/heure
- Besoin de réponses rapides pour ne pas impacter les performances de l'API

Solution avec Redis

- Comptage des requêtes
  - Utilisation de compteurs Redis pour suivre le nombre de requêtes par utilisateur
  - Clé : rate_limit:{user_id}:{window}

        - Incrémentation du compteur à chaque requête
- Fenêtres glissantes
    - Utilisation de fenêtres temporelles glissantes pour un contrôle plus précis
    - Stockage des timestamps des requêtes dans un sorted set
- Vérification des limites
    - Comparaison rapide du compteur avec la limite définie
    - Rejet des requêtes dépassant la limite

Écrivez le code Python qui permet de gérer efficacement et précisément le contrôle du débit des requêtes API en utilisant deux approches différentes :

1. **Compteur simple** : Implémentez un mécanisme qui limite le nombre total de requêtes pouvant être effectuées sur une période fixe (par exemple, 100 requêtes par heure).
2. **Fenêtre glissante** : Implémentez une méthode plus sophistiquée de contrôle du débit utilisant une fenêtre glissante, qui permet d'autoriser un certain nombre de requêtes dans un intervalle de temps dynamique (par exemple, 10 requêtes par minute).

   https://redis.io/glossary/rate-limiting/