BASES DE DONNÉES

Redis

Système de bases de données clé-valeur en mémoire

Cours complet — architecture, structures de données, persistance,
haute disponibilité et cas d'utilisation pratiques

Sommaire

1 Qu'est-ce que Redis ?

REmote DIctionary Server — magasin clé-valeur en mémoire, open source

Redis stocke des données en RAM pour un accès en microsecondes/millisecondes. Contrairement à un cache simple, il propose des structures de données riches : listes, ensembles, tables de hachage, flux (streams), et bien plus.

Vitesse

Données en mémoire vive, latence de l'ordre de la microseconde à la milliseconde.

Exécution atomique

Modèle mono-thread pour les commandes : pas de verrous complexes.

Structures riches

String, Hash, List, Set, Sorted Set, Stream, Bitmap, HyperLogLog, Geo.

Persistance optionnelle

RDB (snapshots) et/ou AOF (journal des écritures).

1 Pourquoi utiliser Redis ?

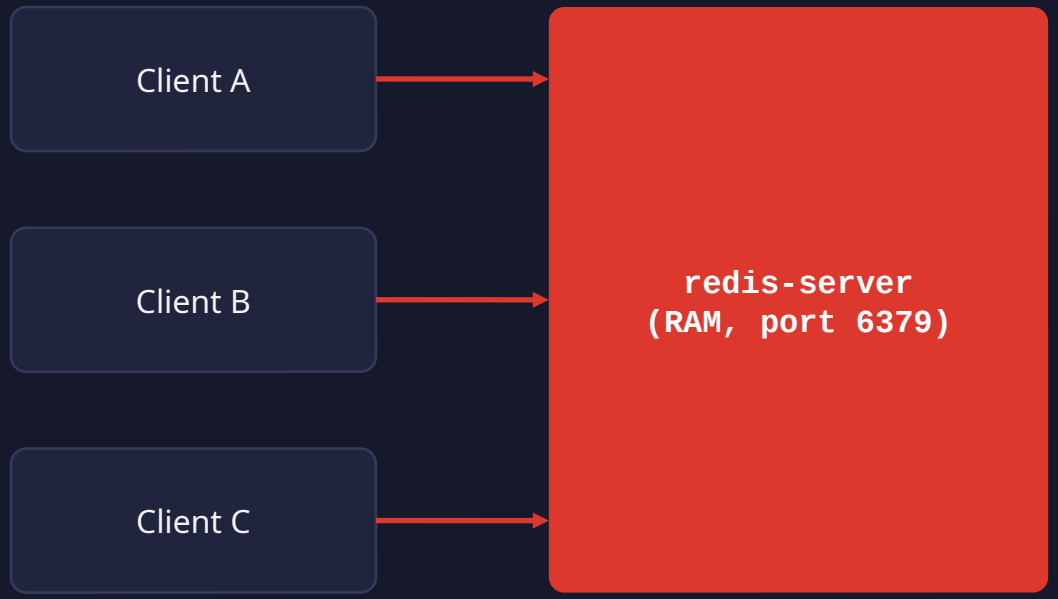
Un besoin métier → une structure Redis adaptée

Besoin	Apport de Redis
Réduire la charge sur une base SQL/NoSQL	Cache devant la base principale
Sessions utilisateurs web	Stockage rapide, TTL natif
Compteurs, votes, statistiques temps réel	Opérations atomiques (INCR, HINCRBY)
Files d'attente / jobs asynchrones	Listes, Streams
Classements (leaderboards)	Sorted Sets
Limitation de débit (rate limiting)	Compteurs + expiration

1 Architecture générale

Modèle client-serveur, protocole RESP

Redis fonctionne en client-serveur : un processus redis-server écoute sur un port (6379 par défaut) et des clients s'y connectent via le protocole texte RESP (REdis Serialization Protocol). Toutes les données tiennent en mémoire vive, avec des mécanismes optionnels de sauvegarde sur disque.



2 Installation et prise en main

Démarrer un serveur Redis en quelques secondes

Docker (recommandé pour tester)

```
docker run -d --name redis-test \
  -p 6379:6379 redis:7-alpine
```

Debian / Ubuntu

```
sudo apt-get install redis-server
sudo systemctl enable --now redis-server
```

Client redis-cli

```
$ redis-cli
127.0.0.1:6379> PING
PONG
127.0.0.1:6379> SET utilisateur:1:nom "Yvan"
OK
127.0.0.1:6379> GET utilisateur:1:nom
"Yvan"
```

Convention de nommage : clés hiérarchiques séparées par « : » → utilisateur:1234:profil

3 Types de base — Strings & Hashes

Les deux structures les plus utilisées

Strings

```
SET compteur:visites 0
INCR compteur:visites          # -> 1
SET session:abc "user_id=42" EX 3600
```

Cas d'usage : compteurs, caches simples, verrous distribués, jetons.

Hashes

```
HSET produit:1001 nom "Clavier" \
  prix 79.90 stock 24
HGET produit:1001 prix
HINCRBY produit:1001 stock -1
```

Cas d'usage : profils utilisateurs, fiches produits, objets métier.

Choisir la bonne structure : un objet à plusieurs champs → Hash plutôt que plusieurs clés String séparées.

3 Types de base — Lists, Sets, Sorted Sets

Séquences, ensembles et classements

Lists

Séquences ordonnées. LPUSH / RPUSH / LRANGE / BLPOP (attente bloquante).

Cas d'usage : files d'attente de jobs, timelines, historique récent.

Sets

Collection unique, opérations ensemblistes. SADD / SINTER / SUNION / SCARD.

Cas d'usage : tags, appartenance à un groupe, déduplication.

Sorted Sets

Comme un Set, avec un score de tri. ZADD / ZINCRBY / ZREVRANGE / ZRANK.

Cas d'usage : leaderboards, files de priorité, planification différée.

4 Types de données avancés

Streams, Bitmaps, HyperLogLog, Geospatial

Streams

Journal append-only avec groupes de consommateurs (XADD, XREADGROUP). Ingestion de capteurs, event sourcing, pipelines temps réel.

Bitmaps

Manipulation bit à bit, très économe en mémoire (SETBIT, BITCOUNT). Suivi de présence quotidienne, feature flags par utilisateur.

HyperLogLog

Estimation de cardinalité avec ~0.81% d'erreur, empreinte fixe ~12 Ko (PFADD, PFCOUNT). Comptage de visiteurs uniques à grande échelle.

Geospatial

Coordonnées géographiques sur des Sorted Sets (GEOADD, GEOSEARCH). Recherche de points d'intérêt, livraison, géolocalisation de flotte.

5 Expiration, éviction et mémoire

TTL natif et politiques d'éviction (maxmemory-policy)

```
SET session:xyz "donnees" EX 1800   # expire dans 30 min
TTL session:xyz                      # temps restant
PERSIST session:xyz                  # annule l'expiration
```

Politique	Comportement
`noeviction`	Refuse les nouvelles écritures (erreur)
`allkeys-lru`	Évince la clé la moins récemment utilisée, toutes clés
`volatile-lru`	Idem, mais seulement parmi les clés avec TTL
`allkeys-lfu`	Évince la clé la moins fréquemment utilisée
`volatile-ttl`	Évince la clé dont le TTL expire le plus tôt

6 Persistance des données

Deux mécanismes pour survivre à un redémarrage

RDB (snapshot)

Photo complète de la base à intervalles définis (save 900 1).

\+ Fichier compact, restauration rapide, bon pour sauvegardes et réplication.
− Perte possible des données depuis le dernier snapshot en cas de crash.

AOF (journal)

Journalise chaque commande d'écriture, rejouée au démarrage (appendfsync everysec).

\+ Durabilité forte (perte max ~1s avec everysec).
− Fichier plus volumineux, restauration plus lente.

En production : combiner RDB (sauvegardes rapides) + AOF (durabilité fine).

7 Pub/Sub et messagerie

Communication temps réel entre clients

```
# Terminal A (abonné)
SUBSCRIBE canal:notifications

# Terminal B (éditeur)
PUBLISH canal:notifications "Nouvelle commande
#4521"
```

À retenir

Pub/Sub est « fire-and-forget » : un message publié sans abonné connecté est perdu. Pour de la messagerie persistante, préférer les Streams (module 4).

Cas d'usage : notifications temps réel, invalidation de cache distribuée, chat.

8 Transactions et scripting

MULTI/EXEC, WATCH et scripts Lua atomiques

Transaction MULTI/EXEC

```
MULTI
INCR compteur:a
INCR compteur:b
EXEC
```

Script Lua atomique

```
local solde = tonumber(redis.call(
  'GET', KEYS[1]))
if solde >= tonumber(ARGV[1]) then
  redis.call('DECRBY', KEYS[1], ARGV[1])
  return 1
else return 0 end
```

Cas d'usage : rate limiting précis, débit atomique de stock, opérations métier multi-étapes garanties atomiques.

9 Réplication, Sentinel, Cluster

Trois niveaux de haute disponibilité et de scalabilité

Réplication

Un maître, plusieurs répliques en lecture seule.

Quand : petite charge, besoin de lecture scalée.

Sentinel

Supervision du maître/répliques, failover automatique en cas de panne.

Quand : haute disponibilité sans besoin de sharding.

Cluster

Partitionnement automatique sur 16384 hash slots, répliques par shard.

Quand : dataset trop volumineux pour un seul nœud.

```
redis-cli --cluster create \
  192.168.1.1:6379 192.168.1.2:6379 192.168.1.3:6379 \
  --cluster-replicas 1
```

10 Sécurité

Quatre réflexes indispensables en production

Authentification

requirepass, ou ACL (ACL SETUSER) depuis Redis 6 pour des permissions fines.

Chiffrement en transit

TLS natif depuis Redis 6.

Isolation réseau

Ne jamais exposer Redis directement sur Internet ; pare-feu / VPC privé.

Commandes dangereuses

Renommer ou désactiver FLUSHALL, CONFIG, KEYS via rename-command.

11 Performance et bonnes pratiques

Sept réflexes pour une instance Redis saine

1. Éviter KEYS * en production → utiliser SCAN (curseur, non bloquant)
2. Pipeliner les commandes pour réduire les allers-retours réseau
3. Choisir la bonne structure (Hash plutôt que plusieurs Strings)
4. Limiter la taille des collections (LTRIM sur les listes de logs)
5. Monitorer avec INFO, SLOWLOG, redis-cli --latency, RedisInsight
6. Dimensionner maxmemory explicitement
7. Utiliser des clés expirables pour tout ce qui est temporaire

12 Cas d'utilisation — 1/2

Cache, sessions et limitation de débit

Cache applicatif

Motif cache-aside : l'application vérifie Redis avant d'interroger la base principale.

redis.get(cle) sinon requête SQL + redis.set(cle, valeur, ex=300)

Sessions web

SET session:id "user_id=42" EX 1800

Partage des sessions entre plusieurs serveurs applicatifs sans état partagé en base SQL.

Rate limiting

INCR sur une clé rate:{utilisateur}, EXPIRE au premier appel, comparaison au seuil autorisé.

Cas d'usage : quotas d'API par utilisateur/minute.

12 Cas d'utilisation — 2/2

Files de tâches, classements et verrous distribués

File de tâches

LPUSH file:emails {...} côté producteur, BLPOP file:emails côté worker (traitement asynchrone).

Cas d'usage : envoi d'emails, traitement d'images.

Leaderboard

ZADD tournoi:s1 4200 equipe:12
ZREVRANGE tournoi:s1 0 4 WITHSCORES

Classement mis à jour et consultable en temps réel.

Verrou distribué

SET verrou NX EX 30 pour acquérir, DEL pour libérer.

Pour un verrou robuste multi-nœuds : algorithme Redlock.

13 Redis face aux alternatives

Choisir le bon outil selon le besoin

Critère	Redis	Memcached	MongoDB	PostgreSQL
Modèle	**Clé-valeur riche**	Clé-valeur simple	Documents JSON	Relationnel
Persistance	**Optionnelle**	Non	Oui	Oui
Structures avancées	**Oui**	Non	Requêtes riches	SQL complet
Cas d'usage typique	**Cache avancé, temps réel**	Cache simple	Documents	Données transactionnelles

Redis excelle en complément d'une base principale : tout ce qui doit être rapide et éphémère (ou semi-durable).

14 Projet pratique de synthèse

Mini système de billetterie d'événement

1	**Modéliser les événements**	Hash evenement:{id} → nom, date, places_dispo
2	**Réserver sans survente**	DECRBY atomique protégé par script Lua
3	**Notifier de façon asynchrone**	Stream pour l'envoi des emails de confirmation
4	**Classer les événements**	Sorted Set des événements les plus réservés
5	**Mettre en cache les pages**	Cache-aside sur le détail événement, TTL 60s
6	**Limiter les abus**	Rate limiting : 5 réservations / minute / utilisateur

Ce projet couvre Strings, Hashes, Sorted Sets, Streams, scripting Lua et rate limiting.

Pour aller plus loin

- **Documentation officielle** — redis.io/docs
- **Exploration interactive** — commande COMMAND en CLI pour toute la syntaxe
- **Interface graphique** — RedisInsight — exploration et monitoring
- **Modules à explorer** — RedisJSON, RedisSearch, RedisTimeSeries, RedisGraph, RedisBloom

Merci — chaque module peut être approfondi en TP séparé.