

## Séance 2

# Modèle clé-valeur Riak, Memcached, Redis



Ce(tte) œuvre est mise à disposition selon les termes de la Licence Creative Commons Attribution – Pas d'Utilisation Commerciale – Pas de Modification 4.0 International.

# Rappels

- Comparaison **modèle relationnel et NoSQL**
  - Historique du relationnel et émergence du NoSQL
  - Propriétés ACID et BASE, théorème CAP
- Caractéristiques des **bases de données NoSQL**
  - Monde du Big Data, Open Data, Google et Amazon
  - Défaut d'impédance, intégration/application, relation/agrégat
  - Schéma implicite, développeur au centre

# Objectifs

- Le modèle **clé-valeur**
  - Principe et caractéristiques du stockage clé-valeur
  - Cas d'utilisation et de non utilisation
  - Modèles de répartition des données
- **Exemples** de bases de données clé-valeur
  - Riak
  - Memcached
  - Redis

# Modèle clé-valeur



# Clé-valeur (1)

- Base de données **clé-valeur** similaire à table de hachage

*Stocke des paires clé-valeur, identifiables par leur clé*

- Similaire à une table relationnelle avec **deux colonnes**

*Utilisé lorsqu'on fait les recherches sur la clé primaire*

- Très bonnes performances grâce à l'**indexation sur la clé**

<b>Id</b>	<b>Name</b>
16139	Alexis
13019	Charles
15027	Sam
10003	Damien

# Clé-valeur (2)

- Le plus simple espace de stockage de type NoSQL

*Point de vue de l'API permettant de l'utiliser*

- Essentiellement **trois opérations** sur le store

*Récupérer/définir une valeur pour une clé, supprimer une clé*



redis



ORACLE  
BERKELEY DB

**Hamster DB**  
embedded database



LEVELDB

# Type de données

- La valeur stockée de **type blob** (*Binary Large OBject*)

*C'est à l'application de gérer les valeurs et leur format*

- Parfois des limites sur la **taille** des valeurs stockées

*Pour des raisons de performances*

- Parfois des **contraintes de domaine** sur les agrégats

*Redis supporte des listes, ensembles et hashes*

# API de base

- Trois opérations de base supportées par tous
  - `get(k)` récupère la valeur  $v$  associée à la clé  $k$
  - `put(k, v)` ajoute la paire  $(k, v)$  dans la base
  - `delete(k)` supprime la paire associée à la clé  $k$
- Le moteur peut proposer des opérations spécifiques

*Redis propose l'union d'ensembles, par exemple*

# Cas d'utilisation

- Stockage des **informations d'une session** pour un site web  
*Identifiant unique de session pratique pour base en clé-valeur*
- **Profils et préférences** d'un utilisateur  
*Utilisateur possède un pseudo unique*
- **Panier d'achats** sur un site de e-commerce  
*Stockage du panier d'achats actuel d'un utilisateur*

# Cas de non utilisation

- Liens à établir entre les données liées à différentes clés

*Suivre des liens entre les données n'est pas évident*

- Sauvegarde de plusieurs clés et échec de certaines

*Pas possible de restaurer les opérations déjà réalisées*

- Pas possible de faire des requêtes sur les valeurs

*Sauf pour certaines bases spécifiques*

# Modèle de distribution



# Modèle de distribution

- Plusieurs modèles possibles pour **exploiter un cluster**  
*Fin du scale up (+ gros serveur) vers du scale out (+ de serveurs)*
- L'unité d'information **agrégat** peut facilement se répartir  
*Granulométrie fine de l'information*
- **Plusieurs raisons** d'utiliser un cluster
  - Pouvoir gérer de plus grandes quantité de données
  - Fournir un plus grand trafic de lecture/écriture
  - Résister à des ralentissements ou panne réseau

# Serveur unique

- Pas de distribution dans la version la plus simple

*Exécution sur une seule machine qui gère les lectures/écritures*

- Solution simple à mettre en œuvre et exploiter

- Facile à gérer pour les opérateurs
- Facilité de raisonnement pour les développeurs d'applications

- Adapté pour les bases de données de type graphe

*Où les opérations à faire sont souvent des agrégations*

# Sharding (1)

- Occupation d'un store avec plusieurs utilisateurs

*Lorsqu'ils accèdent à différentes parties des données*

- Le sharding place les données sur différents serveurs

*Horizontal scalability avec déploiement de plusieurs nœuds*

- Répartition de la charge entre les différents serveurs

*Si les utilisateurs demandent des données différentes*

## Sharding (2)



# Répartition de charge

- Dans l'idéal, la **charge** est bien répartie entre les clients
  - Avec 5 nœuds, chaque nœud gère 20% de la charge*
- Placer sur le même nœud les **données accédées ensemble**
  - Utilisation de l'agrégat comme unité de distribution
  - Utiliser la localisation géographique des données
  - Rassembler les agrégats par probabilité d'accès commun
- Possibilité d'avoir du **sharding automatique**
  - Le moteur gère le sharding et le rebalancing des données*

# RéPLICATION master-slave (1)

- **Données répliquées** sur plusieurs nœuds

*Adapté lorsque plus de lectures que d'écritures*

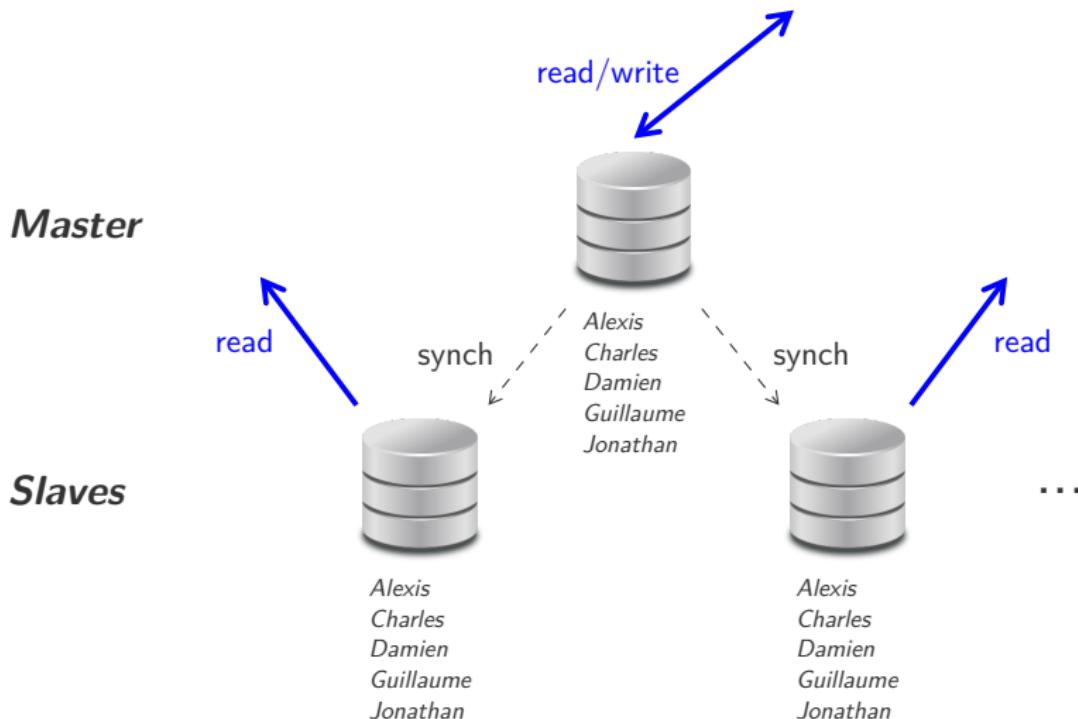
- **Deux types** de nœuds dans le système

- Un nœud master responsable des données et de la mise à jour
- Des nœuds slave qui sont des réplicats du master

- **Deux propriétés** de ce type de réPLICATION

- Read resilience qui permet des lectures si le master tombe
- Valeurs lues par utilisateurs différentes par inconsistance

# RéPLICATION master-slave (2)



# Diffusion des données

- Routage des requêtes en fonction du type

*Read envoyés vers les slaves et write vers le master*

- Synchronisation des slaves par processus de réPLICATION

- Les modifications sur le master sont communiquées aux slaves
- Élection d'un slave comme master si ce dernier tombe

- Deux modes de choix du master

- Choix manuel par configuration
- Choix automatique par élection dynamique

# RéPLICATION peer-to-peer (1)

- **Données répliquées** sur plusieurs nœuds qui sont tous égaux

*Apporte la scalability pour les opérations d'écriture*

- **Synchronisation** de tous les nœuds à chaque écriture

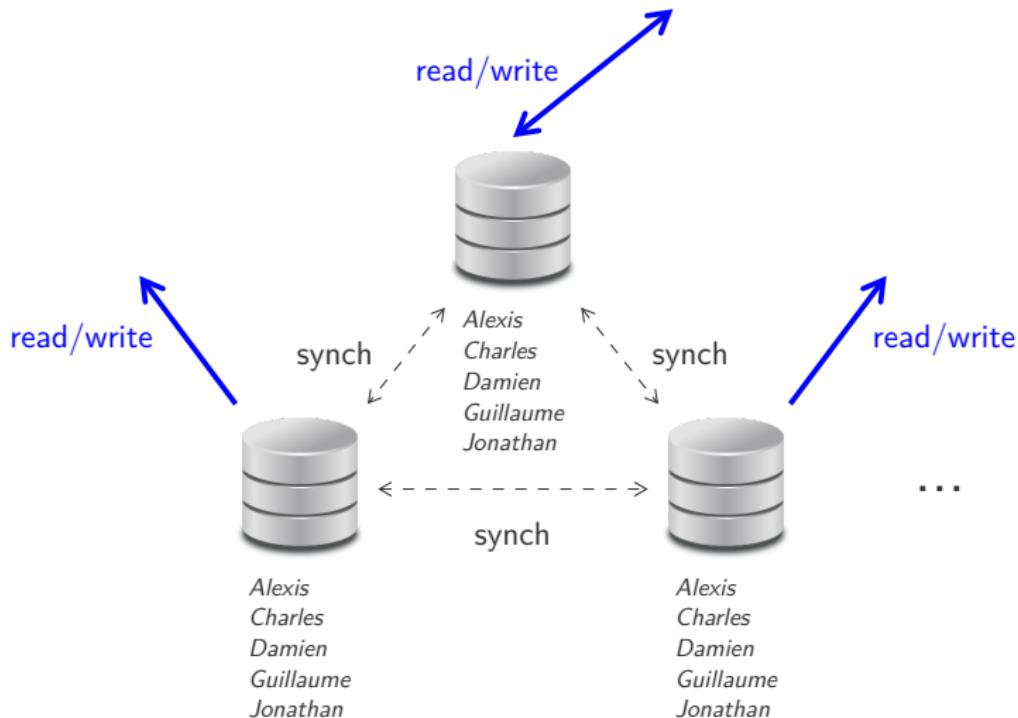
*Conflits d'écriture concurrente, permanente pas comme avec read*

- **Plusieurs propriétés** de ce type de réPLICATION

- Résilience complète en lecture et écriture

- Valeurs lues par utilisateurs différentes par inconsistence

## RéPLICATION peer-to-peer (2)



# Sharding vs replication

- Le **sharding** répartit la charge, pas de résilience

*Données différentes sur les différents nœuds*

- La **réPLICATION** offre de la résilience, lourdeur de synchronisation

*Mêmes données placées sur différents nœuds*

Stratégie	Scaling	Resilience	Inconsistence
Sharding	Write	-	-
RéPLICATION M/S	Read	Read	Oui
RéPLICATION P2P	Read/Write	Read/Write	Oui

# Combinaison sharding/replication

- RéPLICATION **master-slave** et sharding
  - Possibilité d'avoir plusieurs masters, mais un seul par donnée
  - Nœud avec un seul rôle ou rôles mixtes
- RéPLICATION **peer-to-peer** et sharding
  - Données shardées sur des centaines de nœuds
  - Une donnée est répliquée sur  $N$  nœuds (facteur de réPLICATION)



Riak

# Riak

- Crée et développé par la **société Basho**

*Société fondée en 2008 et développe Riak et d'autres solutions*

- Société active avec la dernière version en **novembre 2016**

*Riak est développé en Erlang et la dernière version est Riak 2.2.0*

- Moteur NoSQL **décentralisé** basé sur Amazon Dymano

*Monte en charge avec de nouvelles machines au cluster*

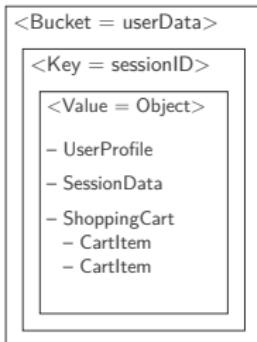
# Bucket

- Riak permet de stocker les clés dans des **buckets**

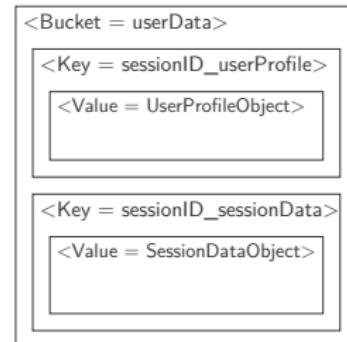
*Agit comme un espace de noms pour les clés*

- Plusieurs possibilités pour exploiter les buckets

*Valeur composée ou séparation en “objets spécifiques”*



*versus*



# Domain bucket

- Domain bucket permet de stocker un type précis de données

*Sérialisation/désérialisation automatique par le client*

- Séparation en buckets pour segmenter les données

- Permet de ne lire que les objets que l'on veut lire
- Permet d'utiliser la même clé à travers les différents buckets

- Lutte contre le défaut d'impédance

*Le store contient directement des objets applicatifs*

# Installation de Riak

- Riak est un programme développé **en Erlang**
- **Plusieurs programmes** proposés après installation
  - riak permet de contrôler les nœuds Riak
  - riak-admin effectue des opérations d'administration

# Lancement d'un nœud

- Lancement d'un nœud Riak avec l'exécutable riak

*Démarrage avec l'option start et arrêt avec stop*

```
& riak start
```

```
& riak ping  
pong
```

# Module Python riak

## ■ Module Python **riak** pour interroger le store

*Ouverture d'une connexion puis méthodes pour l'interroger*

```
1 import riak
2
3 client = riak.RiakClient(protocol='http', http_port=8098)
4
5 print(client.ping())
6 print(client.get_buckets())
```

```
True
[]
```

# Création d'un bucket

- Création d'un **nouveau bucket** avec la méthode `bucket`

*À appeler sur le client Riak*

- Renvoie un objet de **type RiakBucket**

*Permettra d'ajouter et lire des paires clé-valeur*

```
1 import riak
2
3 client = riak.RiakClient(protocol='http', http_port=8098)
4
5 bucket = client.bucket('students')
6 print(bucket)
```

```
<RiakBucket 'students'>
```

# Manipulation de données

- Création d'une **nouvelle donnée** avec la méthode new

*Renvoie un objet RiakObject qu'on va pouvoir stocker*

```
1 import riak
2
3 client = riak.RiakClient(protocol='http', http_port=8098)
4 bucket = client.bucket('students')
5
6 print(bucket.get('16139').data)
7
8 alexis = bucket.new('16139', 'Alexis')
9 alexis.store()
10 print(bucket.get('16139').data)
```

```
None
Alexis
```

# Cluster Riak

- Distribution des données avec un **hashage consistent**
  - Minimise remapping de clés lorsque nombre de nœuds change
  - Réparti bien les données et minimise les hotspots
- Utilisation de **SHA-1** et l'espace des 160 bits comme ring
  - Découpe du ring en partitions appelées “nœuds virtuels”
  - Chaque nœud physique héberge plusieurs vnodes



KNOXVILLE

Memcached

# Memcached

- Système de **cache distribué** à usage général  
*Accélérer site web en cachant des objets en RAM*
- Utilisé en **combinaison** avec une autre base de données  
*Par exemple depuis PHP comme cache vers une base MySQL*
- Memcached est un programme développé **en C**

# Architecture (1)

- Construit sur une architecture **client/serveur**

*Services du serveur exposés sur le port 11211 par défaut*

- Le client effectue des **requêtes par clé** sur la base

*Les clés font 250 octets max et les valeurs jusqu'à 1 Mio*

- Un client connaît **tous les serveurs**

- Les serveurs ne communiquent pas entre eux
- Calcul d'un hash sur la clé pour choisir le serveur

# Architecture (2)

- Données du store sont **stockées en RAM**
  - Plus anciennes valeurs supprimées si à court de RAM
  - Memcached à utiliser comme une mémoire cache transitoire
- Agit comme une grosse **table de hachage**

*On y dépose des paires clé-valeur*

# Module Python memcache

- Module Python `memcache` pour interroger le store

*Ouverture d'une connexion puis méthodes pour les commandes*

```
1 import memcache
2
3 mc = memcache.Client(['127.0.0.1:11211'])
4
5 print(mc.get('13019'))
6 print(mc.set('13019', 'Charles'))
7 print(mc.get('13019'))
8 print(mc.delete('13019'))
9 print(mc.get('13019'))
```

```
None
True
Charles
1
None
```

# L'exemple Trivago

- Trivago utilise Memcached pour sa cache layer  
*Évite une série de requêtes directes vers la DB principale*
- Gros problème soudain avec logs remplis d'**erreurs Memcached**
  - Échec des get et surcharge de la base de données
  - Botnet de plus de 200 pays avec 70K IPs uniques...
  - Saturation interface réseau Memcached au-delà du 1 Gbit/s

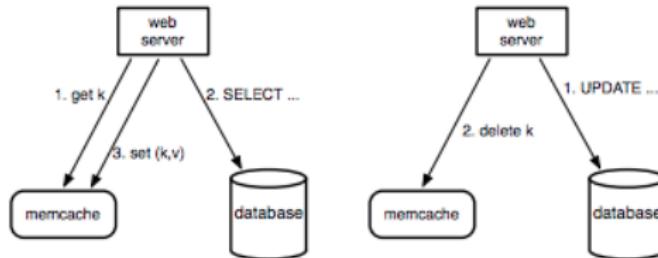
<https://tech.trivago.com/2017/12/19/how-trivago-reduced-memcached-memory-usage-by-50>

# L'exemple Facebook (1)

- Facebook utilise Memcached pour un store distribué

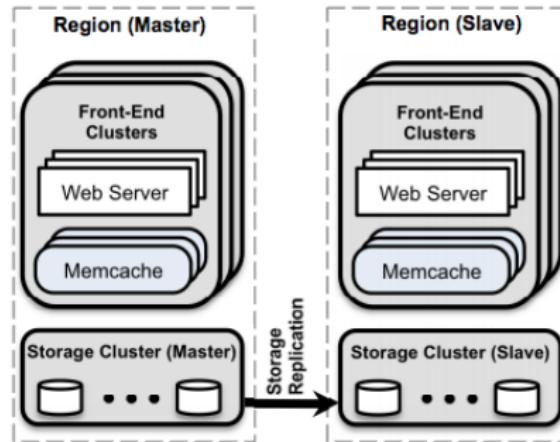
*Dépôt de paires clé-valeurs en mémoire et distribué*

- Deux utilisations différentes pour requêtes ou générique
  - Utilisation comme *demand-filled look-aside cache*
  - Et aussi déploiement d'un store générique distribué



# L'exemple Facebook (2)

- **Pas de coordination** serveur-serveur avec Memcached  
*“Juste” une table de hachage locale en mémoire d'un serveur*
- **RéPLICATION** au sein d'un cluster de serveurs  
*Flux de données depuis le maître vers les esclaves*





# Redis

- Moteur de base de données **en mémoire**

*Manipulation le plus rapide possible de structures de données*

- Joue également le rôle de **cache de données**

*Similaire à Memcached avec un modèle plus riche et solide*

- Restriction sur les **valeurs manipulées**

*Cinq types de valeurs possibles stockées dans la base*

# Type de valeur

- Redis permet de manipuler des **types de données** spécifiques  
*Et ne manipule pas des documents comme d'autres bases*
- **Cinq** différents types de données
  - Chaine de caractères, et valeur numérique ou binaire
  - Liste de chaines (ordre d'insertion maintenu)
  - Ensemble de chaines non trié et sans doublons
  - Hachage (dictionnaire) non hiérarchique
  - Ensemble trié avec association d'une note par élément

# Installation de Redis

- Redis est un programme développé **en C**
- **Plusieurs programmes** proposés après installation
  - `redis-server` permet de démarrer un serveur Redis
  - `redis-cli` est un client en ligne de commande
  - `redis-benchmark` fait un test de performance

# Lancement du serveur

- Lancement du serveur et test de connexion

*Test d'un ping vers le serveur depuis la ligne de commande*

```
& redis-server
```

```
& redis-cli  
127.0.0.1:6379> ping  
PONG
```

# Manipulation de chaines

- Plusieurs commandes de base pour **manipuler les chaines**
  - SET ajoute une nouvelle chaine dans le store
  - GET récupère la valeur associée à une clé
  - DEL supprime une clé du store

```
& redis-cli
127.0.0.1:6379> GET 16139
(nil)
127.0.0.1:6379> SET 16139 "Alexis"
OK
127.0.0.1:6379> GET 16139
"Alexis"
127.0.0.1:6379> DEL 16139
(integer) 1
127.0.0.1:6379> GET 16139
(nil)
```

# Module Python redis

## ■ Module Python `redis` pour interroger le store

*Ouverture d'une connexion puis méthodes pour les commandes*

```
1 import redis
2
3 r = redis.StrictRedis(host='localhost', port=6379, db=0)
4
5 print(r.get('16139'))
6 print(r.set('16139', 'Alexis'))
7 print(r.get('16139'))
8 print(r.delete('16139'))
9 print(r.get('16139'))
```

```
None
True
b'Alexis'
1
None
```

# Manipulation de hachage

- Plusieurs commandes de base pour **manipuler les hachages**
  - HSET ajoute une entrée dans la table de hachage d'une clé
  - HVALS récupère la table de hachage complète d'une clé
  - HGET récupère la valeur d'une entrée d'une table de hachage
  - HDEL supprime une entrée d'une table de hachage

```
& redis-cli
127.0.0.1:6379> HSET 10003 firstname Damien
(integer) 1
127.0.0.1:6379> HSET 10003 favcolour blue
(integer) 1
127.0.0.1:6379> HVALS 10003
1) "Damien"
2) "blue"
127.0.0.1:6379> HGET 10003 favcolour
"Blue"
```

# Équivalence hachage/dictionnaire Python

- Mapping direct entre les hachages et les **dictionnaires Python**

*Initialisation d'un hachage avec hmset*

```
1 import redis
2
3 r = redis.StrictRedis(host='localhost', port=6379, db=0)
4 r.hmset('10003', {
5     'firstname': 'Damien',
6     'favcolour': 'blue'
7 })
8 print(r.dbsize())
9 print(r.hgetall('10003'))
```

```
1
{b'firstname': b'Damien', b'favcolour': b'blue'}
```

# Manipulation de liste

## ■ Plusieurs commandes de base pour manipuler les listes

- LPUSH ajoute une entrée à gauche de la liste
- LPOP retire l'entrée à gauche de la liste
- RPUSH ajoute une entrée à droite de la liste
- RPOP retire l'entrée à droite de la liste
- LRANGE extrait une sous-liste depuis une liste

```
& redis-cli
127.0.0.1:6379> RPUSH students 16139
(integer) 1
127.0.0.1:6379> RPUSH students 13019
(integer) 2
127.0.0.1:6379> LRANGE students 0 -1
1) "16139"
2) "13019"
```

# Équivalence liste/liste Python

- Mapping direct entre les listes et les **listes Python**

*Initialisation d'une liste avec rpush*

```
1 import redis
2
3 data = ['16139', '13019']
4
5 r = redis.StrictRedis(host='localhost', port=6379, db=0)
6 r.delete('students')
7 r.rpush('students', *data)
8
9 data = r.lrange('students', 0, -1)
10 for elem in data:
11     print(elem)
```

```
b'16139'
b'13019'
```

# Persistante des données

- Redis est une **base de données en mémoire** uniquement

*Une fois le serveur quitté, toutes les données sont perdues*

- Possibilité de **sauvegarde régulière** sur disque

*Utilisation du système RDB par défaut, pour snapshots réguliers*

- **Rechargement automatique** de la base de données

*Si un fichier .rdb se trouve dans le bon dossier*

# Expiration

- Possibilité de fixer la **durée de vie** des éléments

*Utilisation de la commande EXPIRE*

- Un élément dans une cache ne devrait **pas vivre pour toujours**

# Exemple de réseau social Redis

- Stockage d'un simple **réseau social** en Redis

*Définition du format des paires clé-valeur à utiliser*

- **Deux types d'objets** à stocker dans le store

- **Utilisateur** a un nom et peut suivre et être suivi par d'autres
  - **Post** est un message, une photo...

- Un utilisateur peut avoir **plusieurs posts**

*Stockage de la liste des posts d'un utilisateur*

# Format des clés (1)

- Définition du **format des clés** à utiliser

*Doit être une simple chaîne de caractères*

- **Convention** pour avoir des clés uniques

- **Utilisateur**

```
user:1:name → Sylvain  
username:Sylvain → 1
```

- **Post**

```
post:1:content → Salut Antoine, je te kiffe !  
post:1:user → 1
```

# Format des clés (2)

- Posts et relations de suivi avec **listes/ensembles**

*Liste de nombres entiers référençant les utilisateurs et posts*

- Utilisation de « **sous-clés** » de user

- **Liste des posts**

`user:1:posts → [3, 2, 1]`

- **Relation de suivi**

`user:1:follows → {2, 3, 4}`

`user:1:followed_by → {3}`

# Identifiant automatique

- Possibilité d'**incrémenter une valeur** avec la commande INCR

*La valeur doit représenter un nombre entier*

- Ajout de deux paires pour représenter les **prochains IDs**

*Clés next\_user\_id et next\_post\_id*

```
1 import redis
2
3 r = redis.StrictRedis(host='localhost', port=6379, db=0)
4 r.set('next_user_id', 0)
5 print(r.get('next_user_id'))
6
7 r.incr('next_user_id')
8 print(r.get('next_user_id'))
```

```
b'0'
b'1'
```

# Créer un nouvel utilisateur

## ■ Définition d'une méthode de **création d'un nouvel utilisateur**

```
1 import redis
2
3 r = redis.StrictRedis(host='localhost', port=6379, db=0)
4 r.set('next_user_id', 0)
5
6 def create_user(username):
7     uid = int(r.get('next_user_id'))
8     r.set('user:{}:name'.format(uid), username)
9     r.set('username:{}'.format(username), uid)
10    r.incr('next_user_id')
11
12 create_user('Alexis')
13 create_user('Damien')
14
15 print(r.get('user:0:name'))
16 print(r.get('user:1:name'))
```

```
b'Alexis'
b'Damien'
```

# Top 5 Redis use cases

- Session **cache** et Full Page Cache (FPC)

*L'avantage de Redis est la persistence*

- Implémentation d'une **message queue** efficace

*Par exemple l'outil Celery pour Distributed Task Queue*

- Développement d'un **leaderboard** avec du comptage

- Exécution de scripts suite à un **événement Pub/Sub**

<https://www.objectrocket.com/blog/how-to/top-5-redis-use-cases/>

# Crédits

- Photos des logos depuis Wikipédia
- <https://www.flickr.com/photos/curioussiow/182224885>
- <https://www.flickr.com/photos/shepherd-distribution-services/5395849861>
- <https://openclipart.org/detail/94723/database-symbol>
- <https://www.flickr.com/photos/heschong/510216272>
- <https://www.flickr.com/photos/dmott9/5662744650>
- <https://www.flickr.com/photos/othree/10945272436>