Conservatoire National des Arts et Métiers
Certificat de spécialisation analyste de données massives

14 juillet 2017
Rapport de projet NFE204

ÉTUDE DE REDIS

Sébastien Gardoll

Base de données documentaires et distribuées
Responsable : Philippe Rigaux

Table des matières

1 Introduction 3

2 Généralités 3
- 2.1 Écosystème et cas d'utilisation . . . 4
- 2.2 Performances annoncées . . . 4
- 2.3 Installation . . . 5
  - 2.3.1 Installation docker . . . 5
  - 2.3.2 Installation par compilation . . . 6
  - 2.3.3 Commandes . . . 6
- 2.4 Clients graphique . . . 6

3 Modèle de données 6

4 Interactions 7
- 4.1 Opérateurs . . . 8
  - 4.1.1 Gestion des paires . . . 8
  - 4.1.2 Gestion de la base . . . 8
  - 4.1.3 String . . . 8
  - 4.1.4 List . . . 8
  - 4.1.5 Hash . . . 9
- 4.2 Discussion . . . 9
- 4.3 Mise en pratique . . . 9

5 Transaction 10

6 Persistance 11
- 6.1 RDB . . . 11
- 6.2 AOF . . . 12
- 6.3 RDB et AOF . . . 12

7 Réplication 12
- 7.1 Principes . . . 12
- 7.2 Mise en pratique . . . 13

8 Partitionnement 14
- 8.1 Modalités . . . 14
- 8.2 Redis-cluster . . . 14
- 8.3 Mise en pratique . . . 15

9 Reprise sur panne (failover) 16
- 9.1 Sentinel et redis-cluster . . . 16
- 9.2 Redis-cluster . . . 16
- 9.3 Élection d'un nouveau maître . . . 17
- 9.4 Reconfiguration dynamique . . . 17

10 Intergiciel orienté message 17

11 Super cache 18

12 Méta-index 18

13 Conclusion **20**

A Version des logiciels **22**

B Listing **22**
B.1 Informations de réplication 22
B.2 Arrêt de redis-slave1 23

C Performances **24**
C.1 Configuration 24
C.2 Benchmark 24

1 Introduction

l'Earth System Grid Federation (ESGF), est une grille de séries temporelles de données scientifiques géolocalisées (températures, précipitations, données biogéochimiques, etc.) au format netcdf. Cette grille met à la disposition de la communauté scientifique internationale (25000 utilisateurs), environ 5 Po de données. Le stockage des données est assuré entre les instituts partenaires (>10 dont la NASA et IS-ENES). Chaque partenaire détient une partie (les données concernant leur région géographique) de l'ensemble des données, les données n'étant pas dupliquées sauf à l'initiative d'un nœud. Les partenaires sont autant de nœuds de la grille ESGF. Ils ont la responsabilité, entre autres, de l'indexation (solr) et de l'éventuelle correction des données qui leurs sont confiées.

Les ingénieurs de la grille souhaitent optimiser le téléchargement des données les plus demandées grâce à un système de cache. Ils souhaitent également la création d'une base de méta-données afin de renforcer la cohérence entre les nœuds de la grille.

Ce rapport présente une étude de redis, un système de base de données NoSQL, à travers les sections 2 à 10, et propose son utilisation comme cache de données (section 11) et comme méta-index (section 12). Enfin, ce rapport se conclut par une synthèse (section 13).

2 Généralités

Redis est un système de bases de données dont les principaux objectifs sont la recherche de performance et de la haute disponibilité dans un environnement distribué (cloud). Redis fait partie de la famille des bases NoSQL.

Redis se présente comme une base de structures de données (data structure store) qui s'appuie fondamentalement sur le concept de paires de clef-valeur. Les structures de données sont des classiques que l'on utilise dans les langages de programmation (liste, dictionnaire, etc.). Elles sont passées en revue à la section 3 et les interactions avec redis, le sont à la section 4. A noter que redis n'impose pas de schéma de données. Ce point sera discuté dans la sous section 4.2.

Sa plus intéressante particularité vient de l'usage de la mémoire RAM (et de la mémoire flash par extension) comme support principal de l'information (in memory data base). La persistance sur disque magnétique est, bien entendu, implémentée mais elle est optionnelle. Dans certains cas, comme la confection d'un cache de données, cette dernière est tout à fait inutile. Les aspects de la persistance des données sont décrits à la section 6.

L'objectif de haute disponibilité de redis est atteint par l'implémentation des mécanismes de réplication et de partitionnement des données, au sein d'un ensemble d'instances redis que l'on appelle redis-cluster. Ces aspects sont respectivement abordés aux sections 6, 8 et 9.

Redis propose également un intergiciel orienté message (publish/subscribe), décrit à la section 10, qui nous intéressera pour l'évaluation de redis comme méta-index.

Enfin, cette section présente son écosystème, ses cas d'utilisation et va également énoncer quelques une de ses performances. Elle se terminera par une courte description de l'installation de redis et une brève présentation de certains clients graphique (GUI) de redis.

2.1 Écosystème et cas d'utilisation

Redis dont le nom est la contraction de REmote DIctionary Server, est un logiciel open source (lien pour le dépôt github), écrit en langage ANSI C, distribué sous licence bsd et disponible sous les systèmes d'exploitation de type *nix (bsd, linux, macosx, solaris, etc.).

A l'origine développé par M. Sanfilippo en 2009, redis est maintenant développé par Redis Labs depuis 2015. Son modèle économique suit le désormais classique logiciel de base en open source et licence libre, d'une part, et outils d'administration, connecteurs pour l'intégration à d'autres plate-formes et support payant, d'autre part. Redis est donc la brique de base de l'offre de Redis Labs qui l'étoffe avec ses modules et ses connecteurs (pour la plupart en licence libre) mais surtout avec son offre payante redis pack (anciennement redis labs enterprise cluster) qui propose aux entreprises tout un écosystème d'outils d'administration et d'analyse pour redis servant à simplifier son déploiement, sa surveillance et assurer une haute disponibilité. Redis Labs propose également une autre offre payante encore plus automatisée avec redis cloud qui promet, à travers une interface web, un déploiement et une administration à porté de clics, sur les quatre grands fournisseurs de clouds à l'heure actuelle (Amazon AWS, Google cloud, Microsoft azure et IBM softlayer).

Les cas d'utilisation de redis annoncés par Redis Labs sont les suivants :

- Analyse de très grands jeux de données
- Moteur de recherche plein texte et données spatialisées
- Transactions à haute vitesse
- Intégration à Spark
- Analyse de séries temporelles
- Cache de données à haute disponibilité

Ces cas d'utilisation s'appuient principalement sur les mécanismes implémentés par redis comme la gestion in memory des données, son modèle de données supportant les données spatialisées et les séries temporelles, ainsi que son mécanisme de transaction. Ces mécanismes seront explicités au cours de ce rapport. Redis Labs met également en avant les performances de redis qui seront décrites à la sous section suivante.

2.2 Performances annoncées

Tirées du site de Redis Labs, voici les facteurs impactant les performances de redis :

- La bande passante réseau et le temps de latence
- Les performances de la RAM, puisque redis est essentiellement une base in memory
- Le cache du CPU, pour la même raison que la RAM. A noter que redis est single threaded, mono fil d'exécution. Il ne tire donc pas partie des processeurs multicoeurs.

Redis déploie quelques outils dont `redis-benchmark` qui automatise une série de tests de performance. Les performances de l'ordinateur sur lequel tourne mon instance de redis sont données à l'annexe C.

Voici quelques exemples de mesures de performances, l'empreinte mémoire de redis et le nombre de requêtes par seconde par nombre de connexions (figure 1) :

To give you a few examples (all obtained using 64-bit instances) :
An empty instance uses ~3MB of memory.
1 Million small Keys -> String Value pairs use ~85MB of memory.
1 Million Keys -> Hash value, representing an object with 5 fields, use ~160 MB of memory.

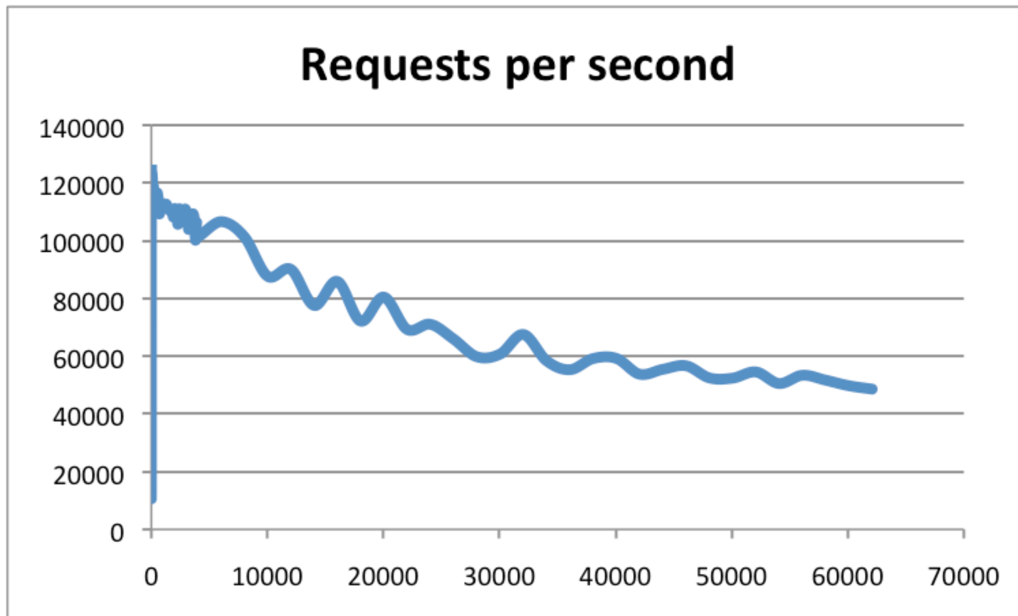

Figure 1 : requêtes par seconde versus nombre de connexions

Dans l'extrait de la documentation de redis, on voit qu'une instance redis prend très peu de place en mémoire (3 MB) et en consomme peu pour stocker des petites paires de clef-valeur. Concernant la réactivité de redis, même par manque de comparaison, on peut affirmer que redis est très performant : 50 000 requêtes par seconde pour un peu plus de 30 000 connexions.

2.3 Installation

Il est relativement facile d'installer redis. Cette sous section explique deux manières différentes : une installation par container docker et une installation par compilation. Quelque soit le mode d'installation, redis ouvre par défaut le port 6379.

2.3.1 Installation docker

C'est la plus facile des deux manières, il suffit de télécharger une image docker et d'instancier un container par la commande suivante :

```
docker run --net host --name redis1 -d redis
```

Où `redis1` est le nom de container et où l'attribution des ports est sans mapping (l'option `--net host`). Nous reviendrons sur les options concernant la gestion d'un cluster redis (réplication, partitionnement, etc.).

Les outils en ligne de commande sont accessibles en se connectant au terminal du container de docker, de la manière suivante :

```
docker exec -it redis1 bash
```

Le client `redis-cli` est accessible directement à l'aide de la ligne suivante :

```
docker exec -it redis1 redis-cli
```

2.3.2 Installation par compilation

Procédure supportée uniquement par les systèmes d'exploitation de type *nix où il s'agit de compiler redis depuis ses sources en exécutant la commande `{make}`(l'installation est limitée au répertoire contenant les sources). Le petit avantage par rapport à l'installation par container réside dans un accès plus facile aux fichiers de configuration de redis, dans un environnement que l'on maîtrise (pas besoin de se connecter au terminal du container).

2.3.3 Commandes

Voici une liste des programmes en ligne de commande déployés par redis (répertoire src pour l'installation par compilation) :

- redis-benchmark : la commande de test de performances
- redis-check-aof et redis-check-rdb : les commandes de contrôle de persistance
- redis-cli : client en ligne de commande
- redis-sentinel : daemon de management de grappes d'instances redis
- redis-server : serveur redis
- redis-trib.rb : script de construction de redis-cluster à partir d'instances de redis

2.4 Clients graphique

Redis est une technologie ayant beaucoup de succès, plusieurs clients graphique lui ont été développé. Voici une sélection de logiciel dont la licence gratuite :

- Fastoredis est un client lourd open source, licence GNU et disponible sur bsd, linux, macosx et windows.
- Redmin est un site internet moyennant une ouverture de compte qui permet de se connecter à ses instances de redis (version gratuite limitée à une seule instance).
- Redis desktop manager est un client lourd open source et licence GNU dont le binaire est gratuit moyennant publicité sous windows et nécessitant une participation financière pour les binaires sous linux et macosx.

3 Modèle de données

Le modèle de données de redis est assez simple. Il s'agit d'une extension au concept de paire de clef-valeur où la clef est un motif binaire et la valeur est une structure de données parmi les suivantes :

- Binary-safe string ou motif binaire
  C'est le type le plus simple pour les valeurs de clef dans redis. C'est également le type des clefs. Cela peut être ce que l'on souhaite comme une image mais il s'agit généralement de chaînes de caractères. Si la clef ou la valeur d'une clef est une chaîne de caractères, il est très important de ne pas oublier que son motif binaire dépend de l'encodage utilisé. Par simplification, nous parlerons de type string, chaîne de caractères ou de motif binaire de façon équivalente.
- List (liste doublement chaînée)
  Structure classique qui est un ensemble ordonné en insertion de motifs binaires. Comme son nom l'indique, l'insertion est soit en tête de liste, soit en queue de liste.
- Set
  Un set est un ensemble non ordonné et sans doublon de motifs binaires.
- Sorted set
  La variante du set, ordonnée selon un entier donné en argument au moment de l'insertion des motifs binaires.
- Hash (dictionnaire)
  Implémentation redis pour un dictionnaire. Le hash est simplement un ensemble de paires de clef-valeur.
- Bit arrays (champ de bits)
  Il s'agit du même type que le binary-safe string. Cependant, ce type permet d'accéder aux bits du motif. Ce type est très utile pour stocker et manipuler un ensemble cohérent de variables binaires.

Le binary-safe string est astuce de redis afin d'éviter l'inférence du code utilisé pour encoder les chaînes de caractères. En effet, redis stocke le motif binaire de la chaîne et non la séquence de caractères qu'il représente. Le problème de codage/décodage des chaînes est laissé au soin des clients de redis. Les clients doivent donc utiliser le même code au risque d'avoir des interprétations différentes des clefs ou des valeurs de clef. De la même façon, redis n'interprétant pas les clefs, c'est aux clients d'établir une convention de nommage des clefs au risque d'écraser les clefs déjà en base. Cependant redis fournit des opérateurs pour vérifier l'existence d'un clef. Redis contraint la taille des motifs binaires à 512 Mo et le nombre de clefs à 2^{32}. Autre particularité redis permet de stocker des paires de clef-valeur de façon temporaire. Lorsque ces paires ont dépassé le temps qu'il leur était alloué, elles sont effacées du système.

Par combinaison de plusieurs paires de clef-valeur, il est possible de modéliser un document comme définit dans le cours NFE204, en stockant comme valeurs dans les structures de données proposées par redis des clefs provenant d'autres paires.

Au prix d'une gestion non automatisée de références à des clefs étrangères, on peut retrouver une forme dégradée de la normalisation des bases relationnelles où la granularité des tables évite la redondance de l'information. L'opération inverse, une simili jointure, est également envisageable. Un exemple est donné à la sous section 4.3.

4 Interactions

Une requête redis a une syntaxe assez simple. Elle est formée d'un opérateur, d'une clef et d'éventuels arguments. L'opérateur et les arguments optionnels dépendent de la structure de données associée à la clef. Si la clef n'existe pas dans la base, redis ne renvoie rien. Si la clef existe mais si l'opérateur utilisé n'est pas adapté au type de structure de données associée à cette clef, redis renvoie une erreur

(WRONGTYPE Operation).

4.1 Opérateurs

Dans tout le reste du rapport, je sous-entendrai l'utilisation du client en ligne de commande. Ce client se connecte par défaut à un serveur local avec le port par défaut (6379). Si l'on souhaite se connecter à un serveur à distance, la syntaxe de son lancement est la suivante :

```
redis-cli --raw -h <hostname> -p <port_number>
```

L'option --raw spécifie à redis-cli d'utiliser le code de caractères du terminal. Voici quelques opérateurs redis (liste complète ici) ; la casse ne compte pas pour le nom des opérateurs) :

4.1.1 Gestion des paires

- EXISTS <clef> : renvoie 1 si la clef est déjà stockée, sinon 0.
- DEL <clef> [<clef>...] : efface les clefs données en argument.
- EXPIRE <clef> <entier> : précise un temps de vie exprimé en secondes.
- TYPE <clef> : renvoie le type de la valeur associée à la clef donnée en argument.

4.1.2 Gestion de la base

- INFO : affiche les informations détaillées de l'instance redis
- ROLE : revoie le rôle de l'instance (master ou slave).
- DBSIZE : renvoie le nombre de clefs dans la base.

Une petite sélection des opérateurs liés au stockage de paires de quelques structures de données. Ces opérateurs sont assez classiques pour les langages de programmation tel que java ou python.

4.1.3 String

- SET <clef> <valeur> : cet opérateur permet de stocker une paire clef-valeur où la valeur est une chaîne de caractères (ou un motif binaire).
- GET <clef> : opérateur permettant de récupérer la valeur de la clef donnée en argument.

4.1.4 List

- LPUSH <clef> <valeur> : ajoute une paire en tête de liste.
- RPUSH <clef> <valeur> : ajoute une paire à la fin de la liste.
- LRANGE <clef> <indice début> <indice fin> : récupère les valeurs dont les indices sont compris dans l'intervalle donné. Si l'indice de début est zéro et l'indice de fin est -1, LRANGE renvoie toutes les valeurs de la liste.

- `LLEN <clef>` : renvoie le nombre d'éléments de la liste.
- `LINDEX <clef> <indice>` : renvoie la valeur à l'indice donné.
- `LPOP <clef>` : renvoie la valeur en tête de liste et l'efface de la liste.

4.1.5 Hash

- `HSET <clef> <champ> <valeur>` : stocke une valeur associée à un champ donné pour la clef précisée en argument.
- `HKEYS <clef>` : renvoie tous les champs pour une clef de hash.
- `HGETALL <clef>` : renvoie tous les champs et leur valeur pour une clef de hash.
- `HGET <clef> <champ>` : renvoie la valeur associée au champ d'une clef de hash, donnés en argument.

4.2 Discussion

Comme énoncé plus tôt, il faut obligatoirement connaître le type de la structure de données associée à la clef pour choisir le bon opérateur sinon le client redis renvoie une erreur. La complexité de la gestion des clefs est donc laissée côté client (heureusement l'opérateur `TYPE` aide). L'utilisation de cette base de données oblige donc à l'autodiscipline concernant le nommage des clefs et le type de structure de données associé aux clefs. Cependant, il est possible de concevoir une règle de nommage de clef particulière pour chaque type de structure.

Par exemple : ”str :1”, ”str :2”, ”str :3”, comme convention de nommage pour les clefs stockant une string ou ”h :1”, ”h :2”, ”h :3”, pour la convention de nommage des clefs liées à des hashes.

4.3 Mise en pratique

Afin d'illustrer l'expressivité du modèle de données de redis et de démontrer les possibilités de simili jointure relationnelle, voici un exemple de modélisation d'une fiche de personnage :

Création de fiches, basiques, de personnages (le retour des requêtes n'est pas représenté) :

```
HSET "h:0" "prenom" "Harry"
HSET "h:0" "nom" "Potter"
HSET "h:0" "date_naissance" "31/07/1980"

HSET "h:1" "prenom" "Ronald"
HSET "h:1" "nom" "Weasley"
HSET "h:1" "date_naissance" "01/03/1980"

HSET "h:2" "prenom" "Hermione"
HSET "h:2" "nom" "Granger"
HSET "h:2" "date_naissance" "19/09/1979"
```

On souhaite ajouter à chaque fiche les amis du personnage. Il s'agit donc de décrire trois ensembles (set) comportant les identifiants/clefs des amis des personnages. Par exemple `"set:0"` est l'ensemble

des amis d'Harry Potter.

```
SADD "set:0" "h:1" "h:2"
SADD "set:1" "h:0" "h:2"
SADD "set:2" "h:0" "h:1"
```

A chaque fiche de personnage, on ajoute un nouveau champ nommé "amis" qui a pour valeur la clef d'un des ensembles d'amis des personnages.

```
HSET "h:0" "amis" "set:0"
HSET "h:1" "amis" "set:1"
HSET "h:2" "amis" "set:2"
```

Pour finir, on simule une requête pourtant sur la fiche de personnage d'Harry Potter, en réalisant à l'aide du dernier opérateur, `SMEMBERS`, une forme de jointure afin d'afficher la clef de la fiche des personnages amis avec Harry.

```
>HGETALL "h:0"
prenom
Harry
nom
Potter
date_naissance
31/07/1980
amis
set:0
>SMEMBERS "set:0"
h:1
h:2
```

Cet exemple montre que redis n'est pas une base orientée document, il est assez lourd de stocker ou d'interroger un document à l'aide d'opérateurs bas niveau. Cependant, il ne faut pas perdre de vue qu'une requête redis est très rapidement exécutée (de complexité souvent constante). Redis simplifie d'ailleurs les requêtes en masse par l'implémentation d'un mécanisme de pipelining. Par exemple :

```
requetes.txt | redis-cli --pipe
```

permet d'enchaîner l'exécution des requêtes contenues dans le fichier `requetes.txt` sans attendre leur acquittement par le serveur redis. D'autre part, redis propose également un interpréteur de script lua (opérateurs `EVAL` et `EVALSHA`). Ces deux mécanismes viennent ajouter une certaine efficacité et expressivité algorithmique nécessaires à l'implémentation de cas d'utilisation réel.

Reste que cet exemple présente un inconvénient important : l'ensemble des requêtes n'est pas exécuté de façon atomique. Les clients sont en concurrence et manipulent les mêmes clefs sans aucune concertation ce qui peut entraîner des incohérences dans l'information stockée dans la base. La section suivante détaille l'utilisation des transactions redis qui sont une réponse à ce problème.

5 Transaction

Redis implémente une forme dégradée de transaction relationnelle qui garantie l'atomicité d'exécution des requêtes qui en font partie. Voici un exemple de transaction redis :

```
MULTI  <-- début de sérialisation des requêtes
# les requêtes parviennent une à une au serveur sans être exécutées
<requête 1>
<requête 2>
<requête 3>
...
EXEC   <-- fin de sérialisation des requêtes et déclenchement de l exécution
 ↪  séquentielle et sans interruption des requêtes reçues
```

Les transactions de redis garantissent seulement l'exécution sans interruption des requêtes reçues entre les deux opérateurs. L'opérateur `DISCARD` permet de vider la file de requêtes reçues par le serveur (avant l'opérateur `EXEC`). Un point important : redis ne stoppe pas l'exécution de la file de requêtes en cas d'erreur (réseau, erreur de programmation, etc.) et n'effectue pas le rollback des requêtes qui ont réussis. C'est la grande différence avec les transactions relationnelles. Le rollback doit être implémenté côté client à l'aide du retour des exécutions des requêtes (opérateur `EXEC`). Ce comportement permet donc à redis de conserver une simplicité d'implémentation et une certaine rapidité d'exécution des requêtes. Reste qu'il est tout à fait possible qu'une requête d'un autre client vienne modifier la base pendant la sérialisation des requêtes (entre le moment du `MULTI` et du `EXEC`). Si les modifications apportées par les autres clients peuvent porter atteinte à la transaction, redis propose l'opérateur `WATCH` (un mécanisme "optimistic locking") qui sert à marquer les clefs critiques d'une transaction. L'exécution de l'entière transaction par l'opérateur `EXEC` ne s'effectuera pas si l'une des clefs marquées est modifiée entre temps. A noter que les scripts sont naturellement des transactions redis.

Les problèmes côté client sont en partie résolus par la notion de transaction. Dans la section suivante, nous allons aborder la gestion de la persistance des données et la reprise sur panne au niveau serveur.

6 Persistance

Redis est une base in memory, le support d'information primaire est donc la RAM. Si certaines utilisations ne nécessitent pas la persistance des données d'un reboot à l'autre, elles ne sont pas majoritaires. Redis propose donc un mécanisme de persistance qui est détaillé dans cette section. Redis propose également un mécanisme d'éviction de données en mémoire tel qu'on le rencontre dans les caches de données (LRU). Cependant, ce mécanisme ne sera pas abordé car sa présentation dépasse le cadre de ce rapport.

Actuellement, redis implémente deux politiques de persistance des données sur disque : Append Only File (AOF) et RDB. Ces deux politiques peuvent être appliquées séparément ou ensemble car elles se complètent.

6.1 RDB

La politique RDB consiste à effectuer périodiquement des sauvegardes de toute la base sur le disque. Les sauvegardes sont les fidèles images de la base au cours du temps, ce qui permet de le remonter en cas de besoin. Autre avantage, les sauvegardes peuvent être stockées dans le cloud pour plus de sûreté. Cependant, la reprise sur panne s'effectuera à partir la dernière sauvegarde, les modifications survenues après la sauvegarde et avant la panne sont donc perdues. D'autre part, persister la base entière sur disque diminue la disponibilité du serveur.

6.2 AOF

La politique AOF consiste à conserver les requêtes dans un fichier qui sera rejoué en cas de panne, au moment du redémarrage de redis. Cette politique est plus fiable que RDB concernant la perte d'information car seules les éventuelles requêtes traitées au moment de la panne sont perdues. Redis garantit la simplification du fichier de log (construction de requêtes équivalentes à plusieurs requêtes originales) afin d'éviter qu'il ne devienne trop gros. Cependant, le fichier de log est limité à l'espace disque disponible et la reprise sur panne est potentiellement longue si le fichier est grand. D'où l'association des deux politiques.

6.3 RDB et AOF

L'utilisation conjointe des deux politiques résout le problème d'obésité du fichier de log. Le fichier de log est donc limité aux requêtes survenues entre deux sauvegardes. En cas de panne, la perte de données est aussi limitée aux éventuelles requêtes en cours et l'on peut toujours remonter le temps à l'aide des différentes sauvegardes. La persistance des données d'une instance de redis peut également être déléguée à l'un de ses réplicas, comme il sera vu à la section suivante.

7 Réplication

Redis suit le schéma classique de réplication avec une topologie maître-esclave et quelques subtilités (lien).

7.1 Principes

Le nœud redis maître accepte les requêtes en lecture et en écriture et peut avoir plusieurs esclaves pour la réplication de ses données (en mémoire). Ses esclaves peuvent accepter des requêtes en lecture mais refusent les requêtes en écriture. Par défaut, la cohérence est à terme. Redis se différencie des bases NoSQL que l'on a vu en cours par la possibilité de cascader les esclaves afin de soulager le maître du poids de la réplication. De plus, redis laisse le choix de la cohérence - forte ou à terme - au client : si le client utilise l'opérateur `WAIT`, associé à un timeout, lors d'une requête en écriture, le client demande explicitement au maître d'attendre le retour de validation de tous ses esclaves.

Le scénario de réplication est assez simple, le maître entretient un flux de réplication avec chacun de ses esclaves : il transmet toutes les requêtes en écriture qu'il reçoit. En cas d'une panne prolongée le nœud esclave demande à son maître toutes ses données. En cas de panne de courte durée, suffisamment courte pour que les requêtes en écriture adressées au maître pendant la durée de la panne ne dépassent pas son buffer de réplication (par défaut 1Mo ; `repl-backlog-size`), l'esclave a la possibilité de se mettre à jour sans demander l'ensemble des données de la base. Ce mécanisme de reprise diminue très fortement la consommation de ressources du côté maître comme du côté esclave. Concernant la gestion des paires avec expiration, la solution est également simple : les esclaves n'effacent rien d'eux même. En effet, le maintient d'une référence temporelle commune sans gigue n'est pas trivial. En laissant le maître effacer les paires périmées et envoyer les requêtes d'effacement à ses esclaves, le système ne court pas le risque d'une incohérence. Au cas où le maître serait indisponible, l'esclave qui serait élu maître, aura pour tâche de vérifier la péremption des paires et envoyer à ses esclaves les éventuels ordres de destructions. La reprise sur panne du maître sera évoquée dans la section 9.

Un déploiement de nœuds redis devient particulièrement intéressant afin de répondre à des contraintes fortes en disponibilité et temps réel. En jouant sur les politiques de persistance des différents type de nœud, il est possible de rendre un cluster redis très réactif. La sous section suivante présente un tel cas où le nœud maître n'est pas sollicité en lecture mais qu'en écriture avec une politique RDB (qui est par défaut, pour faire simple) et ses esclaves seront chargés de répondre aux (supposées très nombreuses) requêtes en lecture et n'auront pas à persister leur données, à l'aide des options `--save ""` qui désactive RDB et `--appendonly no` qui désactive AOF (bien qu'il soit désactivé par défaut).

7.2 Mise en pratique

Les lignes de commandes suivantes déploient trois containers redis : redis-master1 (port 6379), redis-slave1 (6380) qui est l'esclave direct du maître et redis-slave11 (6381) qui est l'esclave de redis-slave1.

```
docker run -d --net host --name redis-master1 redis redis-server --port 6379
docker run -d --net host --name redis-slave1 redis redis-server --port 6380 \
--slaveof 127.0.0.1 6379 --save "" --appendonly no
docker run -d --net host --name redis-slave11 redis redis-server --port 6381 \
--slaveof 127.0.0.1 6380 --save "" --appendonly no
```

La commandes redis `INFO` renvoie beaucoup d'informations sur le nœud. En lui précisant la section `replication`, on obtient les informations spécifiques au rôle du nœud et à la réplication.

```
docker exec -it redis-master1 redis-cli -h 127.0.0.1 -p 6379 INFO replication
docker exec -it redis-master1 redis-cli -h 127.0.0.1 -p 6380 INFO replication
docker exec -it redis-master1 redis-cli -h 127.0.0.1 -p 6381 INFO replication
```

Le résultat in extenso de ces commandes est en annexe B.1. On constate bien les différents rôles des nœuds (champ `role` et la cascade d'esclaves : champ `slave0`).

La requête `SYNC` va nous permettre de visualiser le flux continu de réplication côté redis-slave11 (6381), le dernier maillon de la chaîne de réplication :

```
docker exec -it redis-master1 redis-cli -h 127.0.0.1 -p 6381 SYNC
```

L'expérience est simple, il suffit de faire une requête d'insertion d'une paire auprès du maître (6379) :

```
docker exec -it redis-master1 redis-cli -h 127.0.0.1 -p 6379
SET "str:0" "hello world"
```

Pour voir apparaître dans la fenêtre de connexion de redis-slave11, les requêtes répliquées :

```
Entering slave output mode...  (press Ctrl-C to quit)
SYNC with master, discarding 100 bytes of bulk transfer...
SYNC done. Logging commands from master.
"PING"
"PING"
"SELECT","0"
"set","str:0","hello world"
"PING"
```

Enfin, si l'on arrête redis-slave1 (`docker stop slave1`), on aperçoit bien (listing en annexe B.2), après l'exécution de la requête `INFO replication`, son absence mais on remarque également la non

reconfiguration automatique du cluster : redis-slave11 n'est pas connecté à redis-master1.

8 Partitionnement

Redis est un système remarquablement configurable et débrayable conçu afin de répondre à un grand nombre de cas d'utilisation. En effet, une instance de redis est par défaut autonome, n'a pas de système de partitionnement ni de réplication ni de failover (par exemple, un cache n'a pas besoin de ça). Cependant, à la section précédente, nous avons, par configuration, créer un espace de réplication pour une instance maîtresse. Nous allons également obtenir le partitionnement par configuration. Redis permet plusieurs modes de partitionnement que la sous section 8.1 passent rapidement en revue. Cette section se propose de détailler plus particulièrement la solution redis-cluster (sous section 8.2 ; indépendant de Redis Labs) qui automatise le partitionnement et assure le failover (voir section 9).

8.1 Modalités

Le partitionnement est obtenu par différents agents et la méthode de partitionnement (intervalle, hachage cohérent, etc.) dépend de l'implémentation de ces agents.

- partitionnement côté client : aussi surprenant que cela puisse paraître, redis offre la possibilité de laisser le partitionnement aux clients. Le client, par la méthode qu'il choisit, place les paires sur les nœuds redis, comme il l'entend. Il existe plusieurs implémentations, généralement utilisant le partitionnement par intervalle mais d'autres comme redis-rb et predis implémentent le hachage cohérent.

- partitionnement assisté par proxy : Les clients n'envoient pas directement leurs requêtes aux nœuds redis mais à des mandataires qui supportent le partitionnement. Twemproxy est une implémentation multi méthodes (dont le hachage cohérent).

- routage de requêtes (query routing) : le client envoie sa requête à l'un des nœuds redis choisit au hasard. Si le nœud contacté n'est pas concerné par la requête, ce nœud donne au client l'adresse du bon nœud pour que le client redirige lui même sa requête.

On notera que les modes de partitionnement côté client et par proxy ont l'avantage de délester les serveurs redis de cette charge.

8.2 Redis-cluster

Redis-cluster est une fonctionnalité stable depuis 2015 et propose le partitionnement automatique et le failover. Elle est devenue la méthode préférée. Pour le mode de partitionnement, redis-cluster est un mélange entre routage de requêtes et partitionnement côté client. Pour la méthode, il s'agit d'une variante de hachage cohérent que redis appelle hash slot. Hash slot consiste, comme le hachage cohérent, en une fonction de hachage immuable (crc16) et une table de correspondance entre le résultat de la fonction de hachage et les fragments. Cependant, Hash slot diffère sur l'évolution de sa table de correspondance. Au lieu de représenter un cercle divisé en secteurs de tailles différentes correspondant aux fragments, hash slot divise le domaine de définition du résultat de la fonction de hachage (0 à 16384) en intervalles de même taille (autant qu'il y a de nœuds maîtres). Chaque fragment corresponde à un seul intervalle. L'ajout ou le retrait des nœuds redis se traduit par une restructuration des intervalles et du mapping intervalle - fragment.

8.3 Mise en pratique

La commande redis-trib.rb permet de former un redis-cluster à partir d'instances de redis autonomes en cours de fonctionnement (trois au minimum). Ce script est disponible dans les sources de redis mais pas dans l'actuelle dernière image docker de redis. Il faut donc télécharger les sources de redis. D'autre part, ce script nécessite l'installation d'un interpréteur ruby ainsi qu'une extension de ruby spécifique à redis :

```
yum install ruby.x86_64 # installation de ruby sous linux red hat et clones
gem install redis # installation de l'extension redis (ruby gem)
```

Le lancement des instances de redis doit s'accompagner de l'option `--cluster-enabled yes` qui active les mécanismes propre au redis-cluster :

```
docker run -d --net host --name redis-master1 redis redis-server --port 6379 \
--cluster-enabled yes
docker run -d --net host --name redis-master2 redis redis-server --port 6382 \
--cluster-enabled yes
docker run -d --net host --name redis-master3 redis redis-server --port 6385 \
--cluster-enabled yes
```

Il faut noter que redis-cluster ouvre deux ports : le port spécifié p (ex : 6379) et un autre port $p + 1000$ (16379) qui sert à la coordination entre les nœuds, le cluster bus. L'option `--net port` est obligatoire car le mapping de docker (NAT en général) n'est pas compatible avec redis-cluster.

Par cette ligne de commande, on lie ensemble les instances autonomes de redis en un redis-cluster :

```
./redis-trib.rb create --replicas 0 127.0.0.1:6379 127.0.0.1:6382 \
127.0.0.1:6385
```

La suite d'adresses décrit les nœuds du cluster. l'option `--replicas` précise le nombre d'esclave (ici zéro). `redis-trib.rb` attribue donc automatiquement les rôles (maître et esclave) et effectue entre les nœuds les liens nécessaires (d'où l'absence de l'option `slaveof` au lancement des serveurs redis). Il faut noter que le raccourci `localhost` ne fonctionne pas, il faut spécifier 127.0.0.1 à la place.

L'extrait suivant provient du retour du script `redis-trib.rb` qui montre la correspondance entre les intervalles (slots) et les nœuds :

```
M: 32172d53fdd4f1df0fda355bac811c5312e8ca5c 127.0.0.1:6379
   slots:0-5460 (5461 slots) master
M: be4992bbeba7cdcef3b4bf3c0265cbe4d1c72f0b 127.0.0.1:6382
   slots:5461-10922 (5462 slots) master
M: b2a29cc9dbddf6d260184ae38e25bb3d6adbe98b 127.0.0.1:6385
   slots:10923-16383 (5461 slots) master
```

Ainsi :

- le nœud redis-master1 est lié à l'intervalle [0, 5460]
- le nœud redis-master2 est lié à l'intervalle [5461, 10922]
- le nœud redis-master3 est lié à l'intervalle [10923, 16383]

Afin d'illustrer la redirection du client (à l'aide de l'option `-c`) induit par le mode de partitionnement par routage de requêtes, nous allons nous connecter au nœud redis-master3 (port 6385) et lui envoyer une requête d'insertion :

```
docker exec -it redis-master1 redis-cli -h 127.0.0.1 -p 6385 -c
127.0.0.1:6385> SET "str:1" "bonjour le monde"
-> Redirected to slot [404] located at 127.0.0.1:6379
OK
127.0.0.1:6379>
```

Le crc16 de la clef de la paire produit l'entier 404 qui appartient à l'intervalle du redis-master1 (port 6379). Redis-master3 renvoie, au client, une redirection vers redis master1 (le client finit par se connecter à celui-ci). Redis-cluster offre la possibilité aux clients de demander les informations de partitionnement afin que le système évite de perdre du temps en connexions inutiles. Bien entendu, redis-cluster propose l'équilibrage des fragments par transfert de paires entre les nœuds.

Enfin il faut noter que l'information du partitionnement n'est pas détenu par un nœud en particulier mais par l'ensemble des nœuds redis qui s'échangent cette information par le biais des heartbeats.

9 Reprise sur panne (failover)

Redis propose non pas un mais deux systèmes de reprise sur panne. Le premier apparu est sentinel et le deuxième est redis-cluster (lien pour les spécifications).

9.1 Sentinel et redis-cluster

Ces deux systèmes dont le but est d'assurer une haute disponibilité, présentent à peu près les mêmes fonctionnalités sauf le partitionnement qui n'est pas supporté par sentinel :

- Suveillance de l'état des nœuds redis
- Remplacement automatique des esclaves déconnectés
- Élection automatique d'esclaves en maîtres suite à la déconnexion de leur maître
- Ajout et suppression de nœuds - découverte automatique de la modification du cluster
- Évitement des single points of failure

La différence entre sentinel et redis-cluster vient principalement de leur modèle de fonctionnement : sentinel prévoit des agents afin de superviser un ensemble d'instances redis alors que redis-cluster est un ensemble d'instances redis qui s'autogèrent. Dans son fonctionnement, sentinel est proche de zookeeper : des processus de surveillance à déployer sur une grappe de maîtres et d'esclaves redis. Les sous sections suivantes détaillent uniquement le fonctionnement de redis-cluster.

9.2 Redis-cluster

Redis-cluster implémente les fonctionnalités citées ci-dessus grâce au cluster bus, la connexion que les nœuds maintiennent entre eux et s'appuie sur un algorithme dérivé de l'algorithme de Raft, une alternative de Paxos. Cet algorithme embarque des horloges vectorielles (nommées epoch pour redis) qui garantissent l'ordre total causal des messages envoyés entre nœuds. Les messages sont les heartbeats envoyés à travers le cluster. La charge utile de ces messages comporte un certain nombre d'informations sur la configuration du cluster et un état local des nœuds du cluster (gossip), néces-

saires à la réalisation des fonctionnalités de failover. Il faut noter que la reprise sur panne automatique des nœuds s'accommode bien avec la reprise sur panne manuelle.

9.3 Élection d'un nouveau maître

Redis-cluster ne possédant pas de nœud arbitre, la promotion d'un esclave suite à la déconnexion de son maître est assez intéressante. Les esclaves sans maître initient de façon non concertée une élection. Cette élection consiste, à obtenir une majorité de voies données uniquement par les nœuds maîtres, ces derniers ne pouvant voter qu'une seule fois pour une période calculée de temps et pour l'ensemble des nœuds esclaves privés du même maître. Ce mode de scrutin favorise les esclaves ayant un connexion très réactive avec les autres maîtres, renforçant ainsi la disponibilité du cluster. Si aucun esclave n'obtient une majorité de voies, l'élection est reconduite. Lorsqu'un esclave est promu, ses données replacent celles de son ancien maître (principe du last failover wins).

9.4 Reconfiguration dynamique

Redis-cluster présente également une propriété très intéressante : la reconfiguration dynamique du cluster. Les développeurs de redis ont voulu répondre à des cas d'utilisation réels où les administrateurs souhaitaient que le cluster migrent automatiquement des esclaves dont le maître en a suffisamment pour des maîtres qui n'en ont pas assez, afin d'équilibrer le nombre d'esclaves par maître. Comme cité dans la section 2.2, une instance de redis vide de données a une emprunte mémoire négligeable. Une technique d'administration habituelle pour redis consiste à provisionner le cluster de beaucoup d'instances redis en attente de remplacer des nœuds défectueux.

10 Intergiciel orienté message

Redis propose un système de messagerie calqué sur le modèle publisher/subscriber. Il s'agit du principe d'abonnement à des canaux de communication où les publieurs ne connaissent pas leurs abonnés et vis et versa. Redis utilise pour cela le cluster bus décrit ci-dessus. Cette fonctionnalité est très intéressante pour la coordination entre les nœuds ESGF.

Concrètement les clients se connectent à la même instance autonome redis ou à l'une des instances d'un redis-cluster et publient un message sur un ou plusieurs canaux , en utilisant l'opérateur `PUBLISH`, ou se mettent en attente (bloquante) de messages sur un ou plusieurs canaux, en utilisant l'opérateur `SUBSCRIBE`.

En reprenant les mêmes commandes que la mise en pratique de la sous section 8.3 pour le partitionnement, on connecte un client sur l'instance redis-master3 (port 6382) et on le met en attente de messages sur le canal nommé `channel1`. Enfin on connecte un client sur l'instance redis-master1 (port 6379) et on lui fait publier un message sur le même canal.

client connecté à redis-master1 :

```
docker exec -it redis-master1 redis-cli -h 127.0.0.1 -p 6379
127.0.0.1:6379> PUBLISH channel1 "Olá mundo"
(integer) 0
127.0.0.1:6379>
```

client connecté à redis-master3 :

```
docker exec -it redis-master1 redis-cli -h 127.0.0.1 -p 6382
127.0.0.1:6382> SUBSCRIBE channel1
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "channel1"
3) (integer) 1
1) "message"
2) "channel1"
3) "Olá mundo"
```

11 Super cache

ESGF est une grille dont les nœuds ont la garde d'un ensemble de données scientifiques et offrent les moyens de sélectionner et de télécharger ces données.

L'idée d'un super cache pour un nœud de la grille ESGF est simple : utiliser un redis-cluster comme une concaténation de la mémoire des machines de cette grappe afin de former un cache de données et de l'intercaler entre le front end web (interface implémentée en python, nommée restful api sur la figure 2) et les systèmes de fichiers distants (lustrefs, nfs et sshfs) contenant les données du nœud. L'objectif de cette architecture décrite à la figure 2 est d'augmenter la réactivité des nœuds pour le téléchargement des données (de taille raisonnable) par les clients des nœuds et de diminuer la consommation en bande passante du réseau (les systèmes de fichiers sont distants).

Dans cette architecture, redis-cluster est configuré pour ne persister aucune donnée puisqu'elles le sont déjà dans les systèmes de fichiers. Comme toutes les requêtes passent par le front end web, il lui est possible de mettre en cache les données concernées par les requêtes en lecture et de mettre à jour les données en cache qui sont concernées par des requêtes en écriture. Les données scientifiques possédant un identifiant unique, il est directement utilisé comme clef pour redis.

Comme la mémoire d'un redis-cluster est limitée, le super cache doit procéder à l'éviction des données les moins demandées pour faire place à la mise en cache de nouvelles données. Il s'appuiera sur l' implémentation de redis d'une variante simplifiée l'algorithme Least Recent Used (LRU).

Redis-cluster, les systèmes de fichiers et le font end web forment le data node. L'index node propose un front end de recherche (search api sur la figure 2) sur les données du data node, en procédant à leur indexation (solr). Le résultat d'une requête de recherche est une url pointant un service du front end web du data node.

12 Méta-index

ESGF est une grille internationale, dont chaque nœud est un data node et un index node. Les données de la grille ne sont pas dupliquées, cependant, il arrive que certains nœuds les dupliquent pour des besoins de proximité. Ces copies de données sont à leur tour indexées et proposées en téléchargement. On distingue donc des données originales et des copies de données, situées sur des nœuds différents. Les données présentent parfois des erreurs qui sont corrigées uniquement sur le jeu original de données, créant ainsi une nouvelle version de ces données. Mais ces corrections ne sont pas propagées aux copies de données puisque la distribution des copies de données n'est pas connue. La grille risque

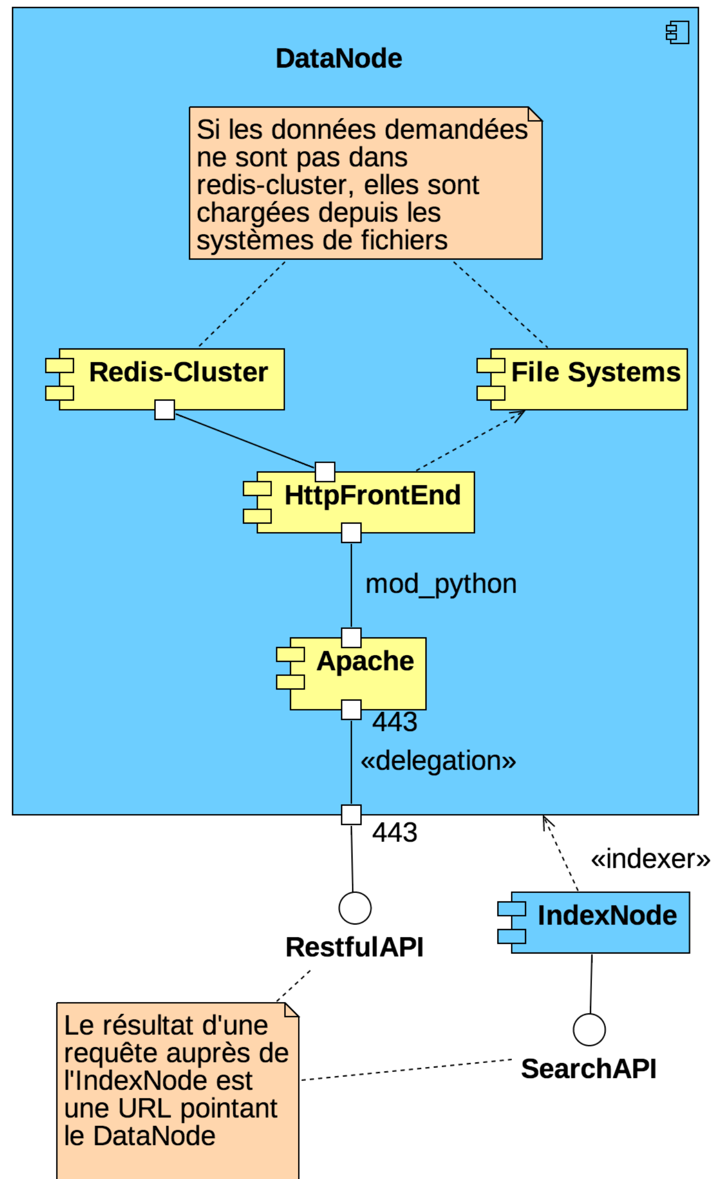

Figure 2 : diagramme de composants UML du super cache

donc une incohérence qu'il est possible de résoudre en créant une base centralisée de méta-données qui contiendra, entre autre, une liste des versions existantes des données.

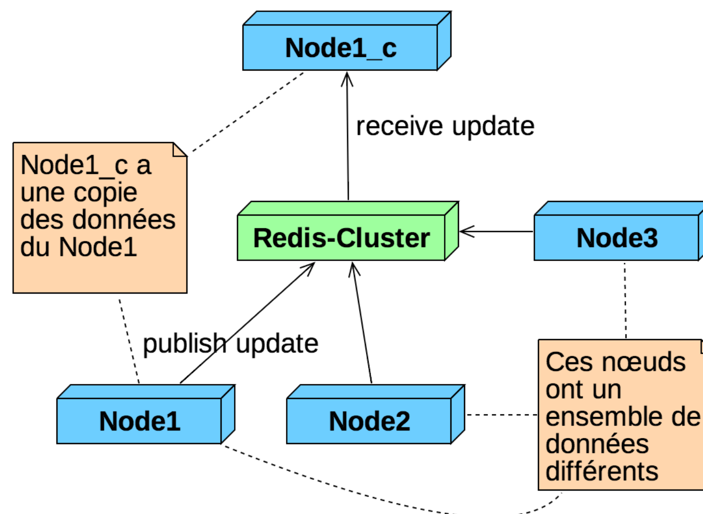

Figure 3 : diagramme de déploiement UML du méta-index

Le diagramme de déploiement UML (figure 3) modélise une solution utilisant redis. Cette solution permet de partager les méta-données entre tous les nœuds ESGF mais également de propager par le biais de messages, l'annonce d'éventuelles mises à jour de méta-données en utilisant l'intergiciel de redis.

Le scénario pour la création d'une nouvelle version de donnée est le suivant : Node1 et Node1_c ont la même version d'un ensemble de données, Node1 détient les données originales et Node1_c a une copie de ces données. Une donnée de Node1 est corrigée et Node1 modifie dans redis les méta-données concernant cette donnée. Il publie également sur le canal lié à cette donnée un message annonçant la mise à jour des méta-données de cette donnée. Node1_c qui est abonné au même canal, reçois le message et récupère les nouvelles méta-données, auprès de redis. Node1_c peut alors se mettre à jour, et par exemple indiquer à ses utilisateurs, lorsque ces derniers font une recherche sur la donnée en question, que Node1_c n'a pas la dernière version de la donnée mais une ancienne version.

13 Conclusion

Redis est une base de données NoSQL qui est populaire au moment de la rédaction de ce rapport. Elle est soutenu par une communauté active et s'accompagne d'un bon nombre d'outils. Cette base est conçue pour stocker des petites structures de données en mémoire mais n'est pas adaptée pour le stockage de documents. Les objectifs des concepteurs de redis sont clairement la performance en lecture et écriture obtenue par son caractère in memory et la disponibilité par l'implémentation du redis-cluster (réplication, partitionnement et failover). Redis propose également un mécanisme de messagerie. Ces caractéristiques font de redis un élément central dans la solution d'amélioration des performances de distribution des données scientifiques de la grille ESGF ainsi que la solution du problème de cohérence entre les copies de ces données.

ANNEXES

A Version des logiciels

- Système d'exploitation : linux centos 7.3
- Docker : 1.12.6
- Redis : 3.2.9

B Listing

B.1 Informations de réplication

```
# MASTER
docker exec -it redis-master1 redis-cli -h 127.0.0.1 -p 6379 INFO replication
# Replication
role:master
connected_slaves:1
slave0:ip=::1,port=6380,state=online,offset=668,lag=0
master_repl_offset:668
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:2
repl_backlog_histlen:667

# SLAVE1
docker exec -it redis-master1 redis-cli -h 127.0.0.1 -p 6380 INFO replication
# Replication
role:slave
master_host:127.0.0.1
master_port:6379
master_link_status:up
master_last_io_seconds_ago:6
master_sync_in_progress:0
slave_repl_offset:696
slave_priority:100
slave_read_only:1
connected_slaves:1
slave0:ip=::1,port=6381,state=online,offset=612,lag=0
master_repl_offset:612
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:2
repl_backlog_histlen:611

#SLAVE11
docker exec -it redis-master1 redis-cli -h 127.0.0.1 -p 6381 INFO replication
# Replication
role:slave
master_host:127.0.0.1
```

```
master_port:6380
master_link_status:up
master_last_io_seconds_ago:5
master_sync_in_progress:0
slave_repl_offset:668
slave_priority:100
slave_read_only:1
connected_slaves:0
master_repl_offset:416
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:2
repl_backlog_histlen:415
```

B.2 Arrêt de redis-slave1

```
# MASTER
docker exec -it redis-master1 redis-cli -h 127.0.0.1 -p 6379 INFO replication
# Replication
role:master
connected_slaves:0
master_repl_offset:1004
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:2
repl_backlog_histlen:1003

#SLAVE11
docker exec -it redis-master1 redis-cli -h 127.0.0.1 -p 6381 INFO replication
# Replication
role:slave
master_host:127.0.0.1
master_port:6380
master_link_status:down
master_last_io_seconds_ago:-1
master_sync_in_progress:0
slave_repl_offset:864
master_link_down_since_seconds:21
slave_priority:100
slave_read_only:1
connected_slaves:0
master_repl_offset:640
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:2
repl_backlog_histlen:639
```

C Performances

C.1 Configuration

- PC AMD FX 8320E : 8 cœurs à 3,2 GHz
- Disque dur SSD : Samsung SSD 850 PRO

C.2 Benchmark

```
./redis-benchmark -q -n 100000
PING_INLINE: 68870.52 requests per second
PING_BULK: 77760.50 requests per second
SET: 67249.50 requests per second
GET: 77279.75 requests per second
INCR: 78369.91 requests per second
LPUSH: 78926.60 requests per second
RPUSH: 78431.38 requests per second
LPOP: 73421.44 requests per second
RPOP: 72046.11 requests per second
SADD: 78247.26 requests per second
SPOP: 78369.91 requests per second
LPUSH (needed to benchmark LRANGE): 79302.14 requests per second
LRANGE_100 (first 100 elements): 35816.62 requests per second
LRANGE_300 (first 300 elements): 17614.94 requests per second
LRANGE_500 (first 450 elements): 12367.05 requests per second
LRANGE_600 (first 600 elements): 10090.82 requests per second
MSET (10 keys): 65832.78 requests per second
```