Mise à l’échelle

Plan et objectifs

- Mise à l’échelle
  - verticale
  - horizontale
- Balanceur de charge
- Mise à l’échelle manuelle et automatique
- Cache
- Réseau de diffusion de contenu
- Mise à l’échelle d’une base de données
- Stratégies de partitionnement

Mise à l’échelle (scaling)

La mise à l’échelle consiste à augmenter la capacité d’un système pour qu’il puisse répondre adéquatement à une plus grande demande.

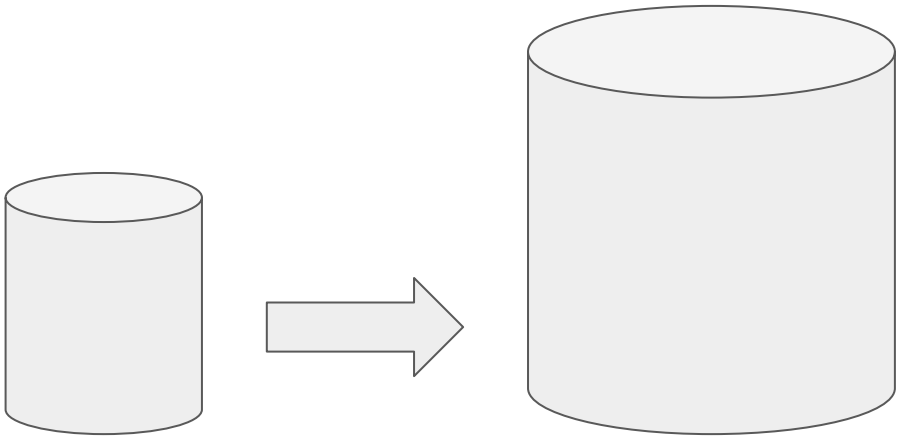
Habituellement, c’est en réponse à une croissance du nombre d’utilisateurs, mais ce peut aussi être requis suivant de nouvelles fonctionnalités qui requièrent plus de ressources.

Mise à l’échelle verticale

La mise à l’échelle verticale consiste à augmenter la capacité de chacun des éléments existants du système.

Par exemple, on peut augmenter la puissance des CPUs, ajouter de la RAM, etc.

Il est aussi possible d’augmenter la capacité en optimisant l’utilisation des ressources (ex: optimisation de code, compression de données, etc).

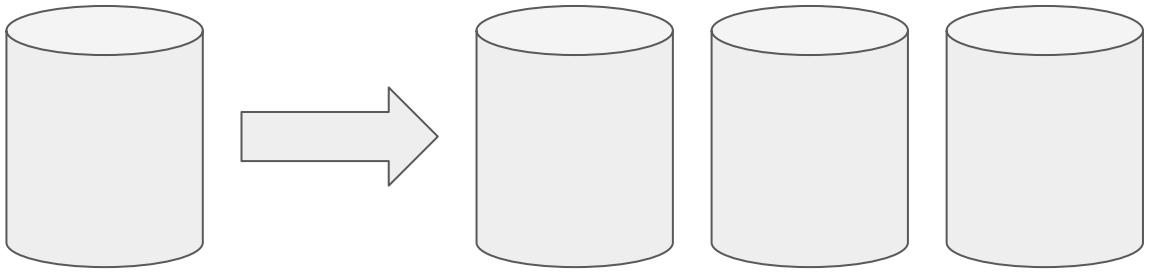


Mise à l’échelle horizontale

Une autre façon d’ajouter de la capacité est d’ajouter des nouvelles instances de composants (serveurs ou conteneurs) qui se partagent la charge.

Ce type de mise à l’échelle rend aussi le système plus résilient, mais plus difficile à opérer et surveiller.

Pour distribuer la charge équitablement, on ajoute un balanceur de charge (load balancer).



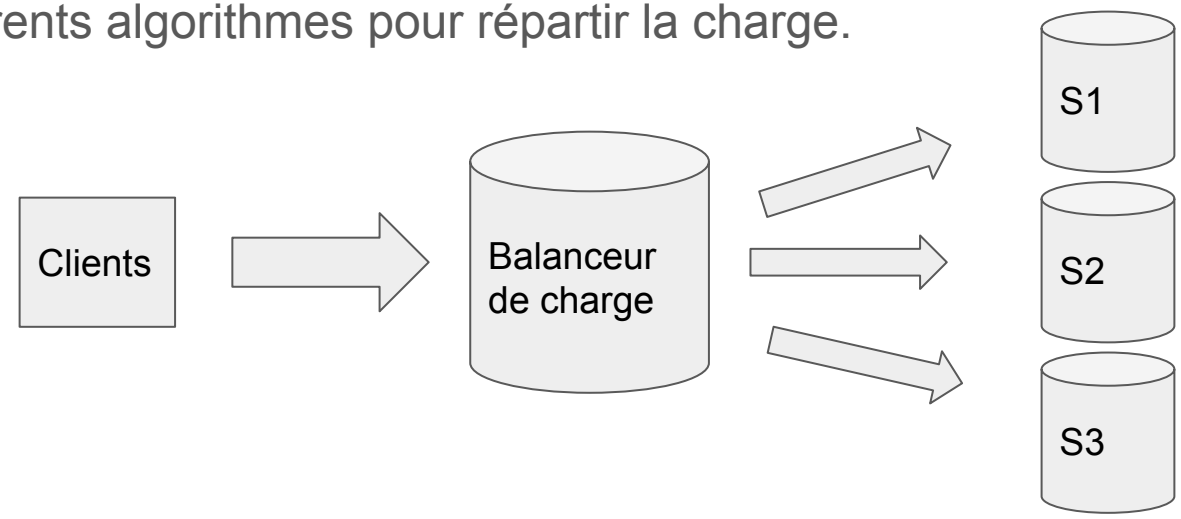
Balanceur de charge

Le rôle du balanceur de charge est de **répartir la charge entre différents composants (serveurs, conteneurs) qui ont un rôle identique** (ex: un service).

Les clients contactent le balanceur de charge et non les instances individuelles. Ceci permet de modifier dynamiquement le nombre d'instances sans avoir à modifier la configuration des clients.

Le balanceur de charge peut être matériel ou logiciel.

Il existe différents algorithmes pour répartir la charge.



Algorithmes de distribution de charge

Planification circulaire (round robin): Les requêtes sont attribuées à tour de rôle à chaque instance. Cette méthode simple fonctionne bien si tous les serveurs (ou conteneurs) ont des caractéristiques identiques.

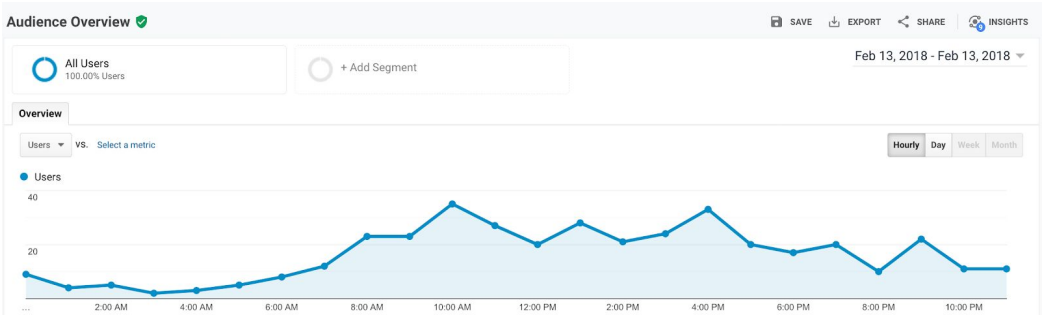
Planification circulaire pondérée (weighted round robin): Un poids est associé à chacun des serveurs (en fonction de leurs caractéristiques). Le balanceur de charge assignera les requêtes proportionnellement au poids de chacun des serveurs.

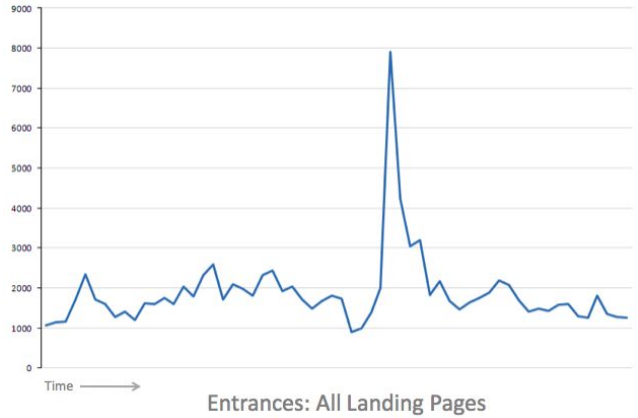
Moins de connexions: La requête est assignée au serveur qui a le moins de connexions.

Certains balanceurs de charge ajustent dynamiquement la répartition en fonction des serveurs qui sont opérationnels. Si un serveur tombe en panne, le balanceur n’assigne plus aucun trafic à ce serveur. Inversement, si un serveur devient actif, le balancer l’ajoute à sa liste (voir health check).

Mise à l’échelle manuelle

En utilisant les métriques d'un système, il est possible de définir un modèle opérationnel qui permet de prévoir à quel moment une mise à l’échelle sera nécessaire en fonction des prévisions de croissance de la base d’utilisateurs.





Avec les solutions d’hébergement en infonuagique, il existe aussi différentes options de mise à l’échelle automatisée.

Mise à l’échelle automatique

Lors de la définition d’un service déployé sur un solution infonuagique, il est parfois possible de spécifier le nombre minimum et maximum d’instances.

Le fournisseur de service s’assurera que toutes les requêtes pourront être traitées en faisant varier le nombre d’instances jusqu’au maximum configuré.

Pour limiter les coûts, le nombre d’instances diminue lorsque la charge diminue.

Pour les solutions plus avancées, il est possible de contrôler la ou les métriques qui déclencheront la mise à l’échelle automatique (ex charge, CPU, etc).

Cache

Un mécanisme très efficace pour améliorer le temps de réponse et diminuer la charge sur un système consiste à conserver en mémoire (accès rapide) le résultat d'opérations coûteuses dont le résultat reste valide pour un certain temps.

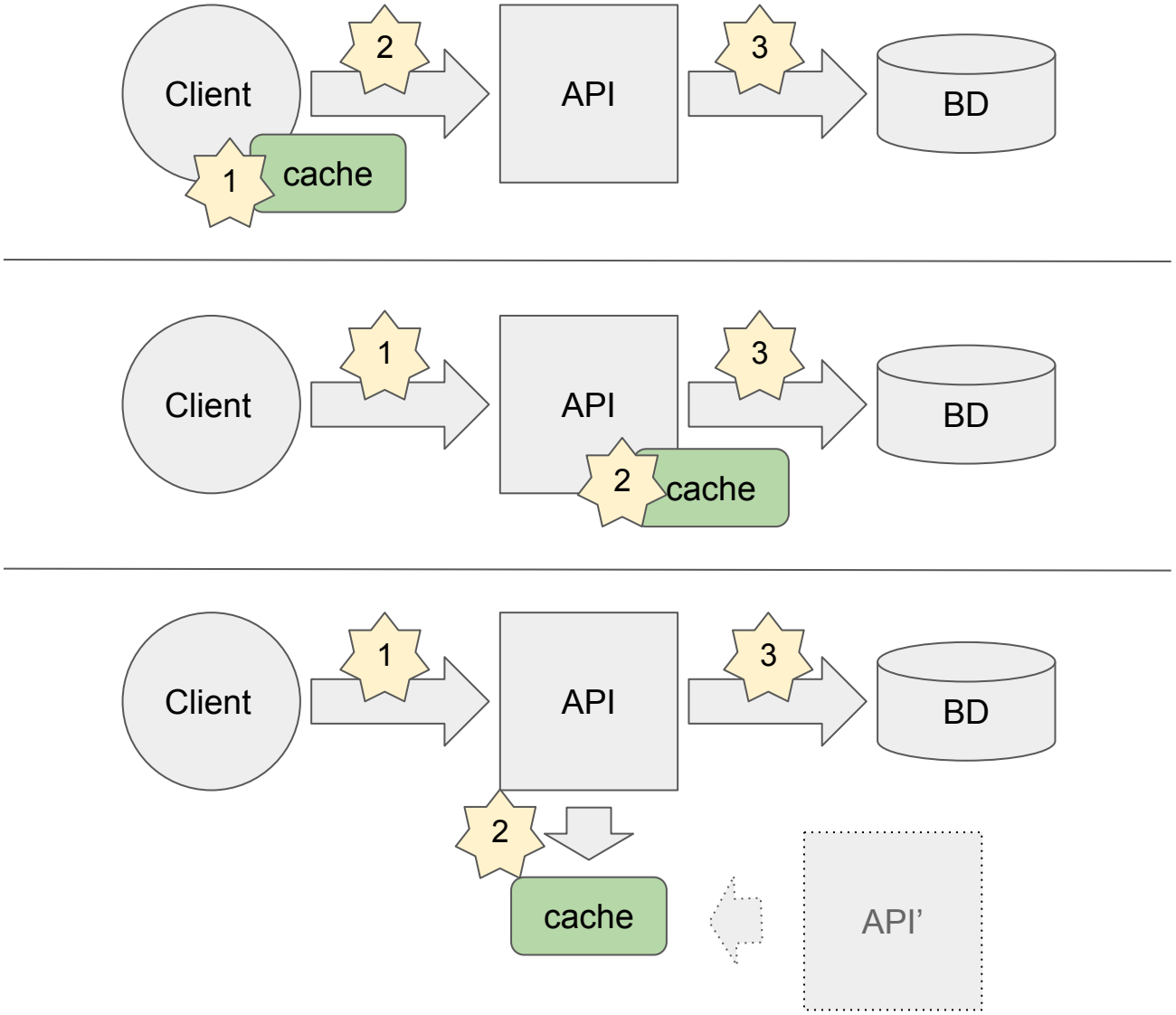
Ce peut être fait dans la mémoire même du processus qui effectue l'opération, dans le composant qui fait la requête ou dans un composant différent.

Par exemple, on pourrait garder les résultats d'un appel d'API de type GET pour éviter de faire un appel à la base de données.

Habituellement, la cache est un dictionnaire *clé - valeur*. Il est possible d'y stocker des valeurs simples (un enregistrement) ou dérivées (résultat d'une opération). Il est important de limiter l'espace utilisé par la cache en utilisant une stratégie du *moins récemment utilisé (LRU)*.

Plusieurs options sont disponibles. Par exemple, pour une cache distribuée, *memcached* et *redis* sont deux options populaires.

Cache - Exemples



Invalider ou mettre à jour

Il existe quelques stratégies pour s'assurer que les données en cache sont valides. Il est possible d'utiliser différentes stratégies pour différentes clés.

- **Invalidation sur écriture** : L'entrée en cache est invalidée (retirée) lors d'une écriture. On doit connaître l'ensemble des clés à invalider en lien avec l' écriture.

- **Remplacement sur écriture**: L'entrée en cache est remplacée lors d'une écriture. Efficace pour les données simples (enregistrement), mais peut être coûteuse si la valeur en cache est le résultat d'une opération.

- **Expiration**: L'entrée en cache est invalidée après un certain temps. C'est la stratégie la plus simple, mais possible seulement dans certains scénarios où on peut permettre qu'une valeur ne soit pas à jour pour un certain temps.

RDC - Réseau de diffusion de contenu (*CDN*)

Un réseau de diffusion de contenu est un ensemble de plusieurs serveurs stratégiquement distribués géographiquement.

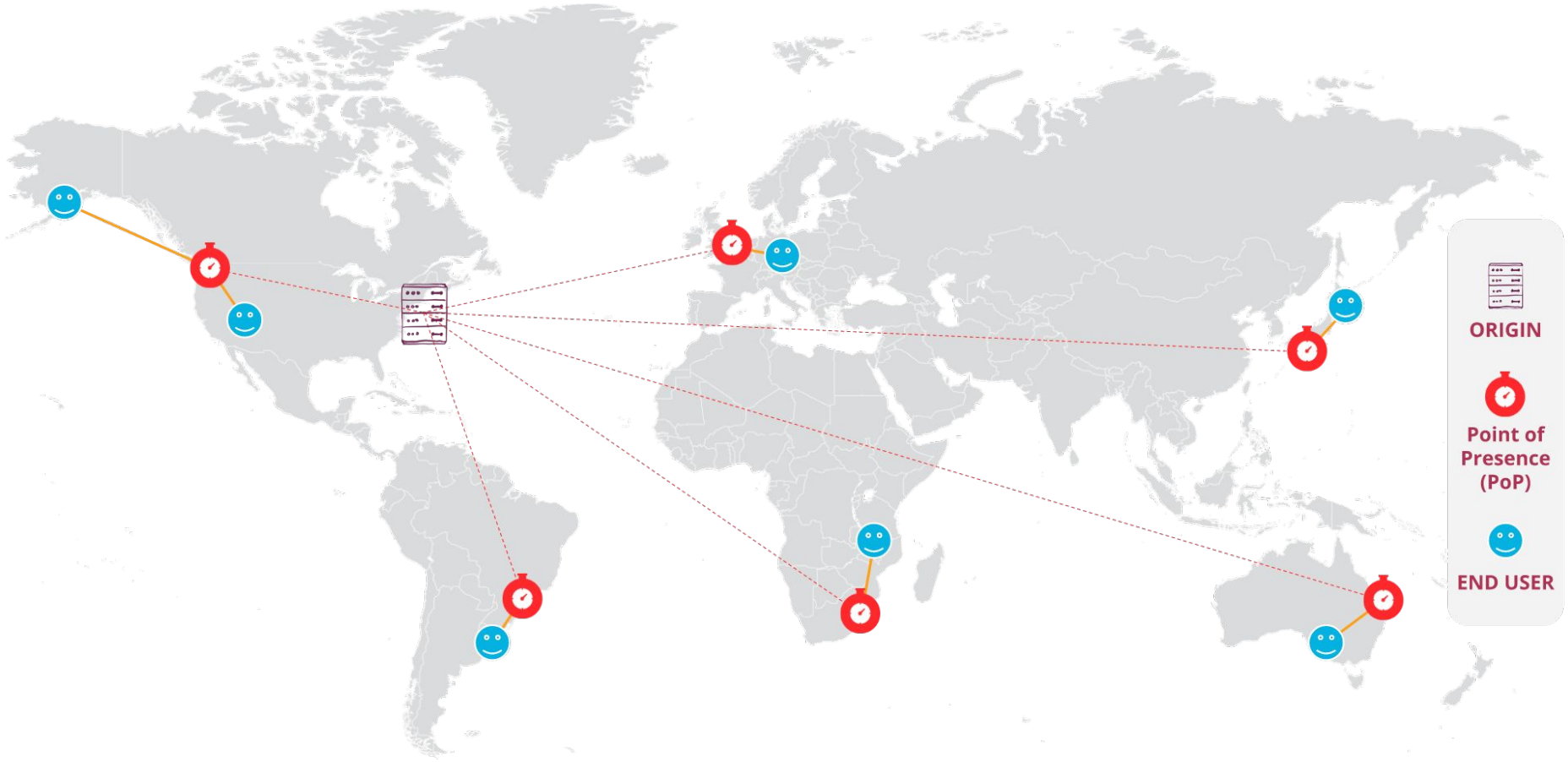
Le serveur d'origine permet de gérer le contenu statique (images, fichiers, scripts, etc) à distribuer.

Les serveurs *points de présence* se synchronisent avec le serveur d'origine.

Lors d'une requête pour le contenu statique, l'utilisateur sera dirigé vers le serveur *point de présence* le plus près, ce qui diminue le temps de chargement.

Comme les RDC demandent une infrastructure élaborée et distribuée à travers le monde, c'est un service que seules certaines entreprises peuvent fournir et gérer.

RDC



src: https://www.fastly.com/learning/what-is-a-cdn

Base de données et mise à l’échelle

Jusqu’à maintenant nous avons surtout considéré la mise à l’échelle horizontale de composants dits *sans état*.

Pour ce type de composant, il suffit d’ajouter des instances pour augmenter la capacité car chaque instance est identique.

Pour une base de données, lorsqu’on atteint une limite (CPU, espace disque, taille de table) on ne peut pas seulement ajouter des nouveaux serveurs, il faut considérer comment les données existantes seront partagées.

Il existe plusieurs stratégies. Nous en examinerons deux:

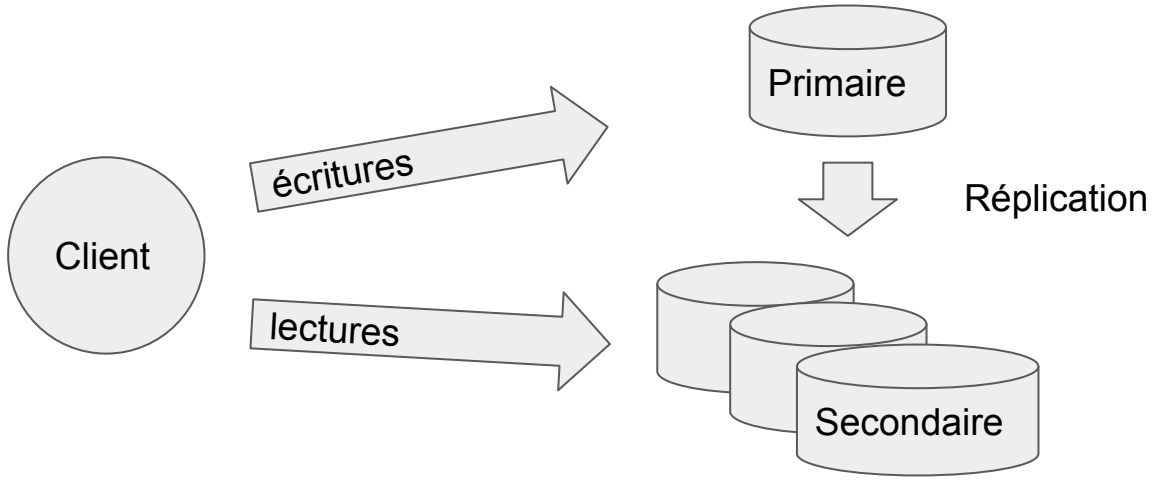
- Réplicats de lecture
- Partitionnement

Réplicats de lecture

Si la limitation de la base de données est liée aux nombres d'appels en lecture, il est possible d'ajouter des réplicats dédiés aux opérations de lecture.

Une serveur devient le serveur primaire et les autres, les serveurs secondaires.

Les opérations d'écriture sont seulement effectuées sur le serveur primaire. Les données sont répliquées vers les serveurs secondaires qui se partagent la charge d'opérations de lecture.



Partitionnement

Si la limitation du serveur de base de données est lié à la quantité de données, il est possible de partitionner (diviser) les données pour ensuite les partager sur plusieurs serveurs.

L'algorithme de partitionnement doit être déterministe: à partir d'une information (clé, valeur d'un champ) on doit être en mesure de retrouver la partition.

Exemple MySQL, création d'une table avec 10 partitions:

```
CREATE TABLE userslogs (
    username VARCHAR(20) NOT NULL,
    logdata BLOB NOT NULL,
    created DATETIME NOT NULL,
    PRIMARY KEY(username, created)
)
PARTITION BY HASH( TO_DAYS(created) )
PARTITIONS 10;
```

Types de partitionnement

Il existe différentes stratégies de partitionnement:

- **Intervalle**: On choisit un champ et on assigne les partitions en fonction d'intervalle de valeur.
  - Par exemple: P1: année < 2015, P2: année = 2016, …, Pn: année > 2022
- **Liste**: On choisit un champ et on assigne les partitions selon la valeur du champ.
  - Par exemple: P1: catégorie = [A, B, C], P2: catégories = [D, F]
- **Clé**: On utilise la clé de l'enregistrement pour partitionner la table en divisant l'ensemble des clés possibles en partitions de tailles équivalentes.
  - Par exemple: clé % nb partitions
- **Hash**: On applique une fonction de hachage sur un ou plusieurs champ pour déterminer la partition (les champs doivent être connus du client!).
  - Par exemple: hash(nom) % nb partitions

Une bonne stratégie de partitionnement distribue les données de façon équitable entre les différentes partitions pour éviter un débalancement en termes de taille et d'utilisation.

Partitionnement et évolution

À mesure que le système évolue, la quantité de données et la charge risquent aussi d'évoluer.

Ainsi, le nombre de partitions créées initialement pourrait s'avérer insuffisant.

Selon le type de partitionnement choisi, il est possible que d'ajouter une nouvelle partition requiert de réassigner des clés (et de transférer beaucoup de données).

Pour un algorithme du type *hash(k) % nb partitions*, augmenter le nombre de partitions engendre beaucoup de ré-assignations.

Exemple

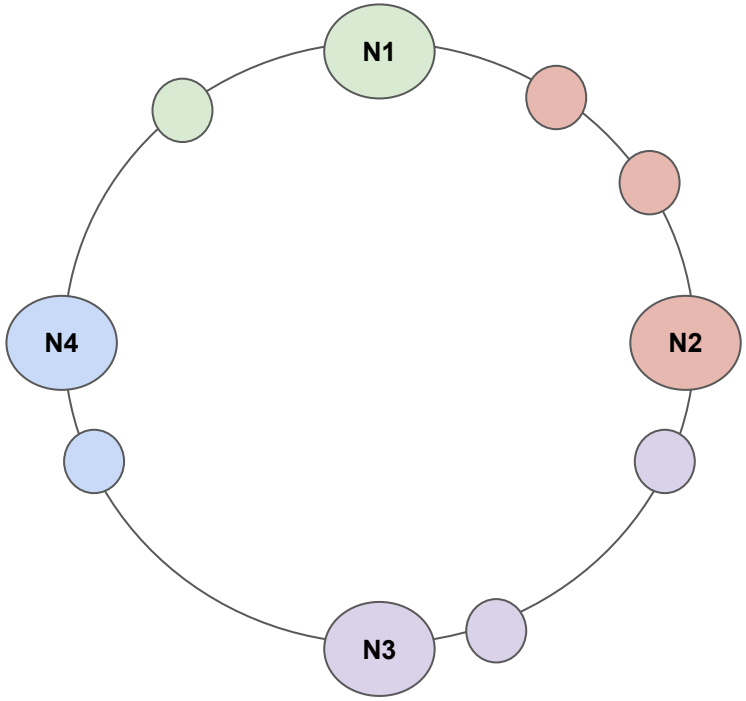
id%2 => 0:0, 1:1, 2:0, 3:1, 4:0, 5:1, 6:0, 7:1, 8:0, 9:1, 10: 0, 11: 1, 12:0

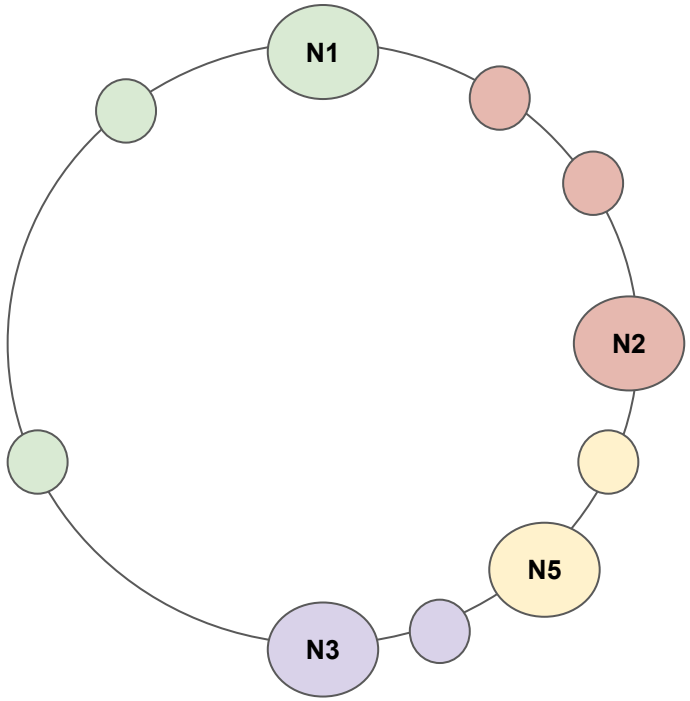
id%3 => 0:0, 1:1, **2:2, 3:0, 4:1, 5:2**, 6:0, 7:1, **8:2, 9:0, 10:1, 11:2**, 12:0

Hachage cohérent

*L'idée derrière l'algorithme de hachage cohérent est d'**associer chaque machine à un ou plusieurs intervalles où les frontières des intervalles sont déterminées en calculant le hash de chaque identifiant**.*

https://fr.wikipedia.org/wiki/Hachage_coh%C3%A9rent





Partitionnement et résilience

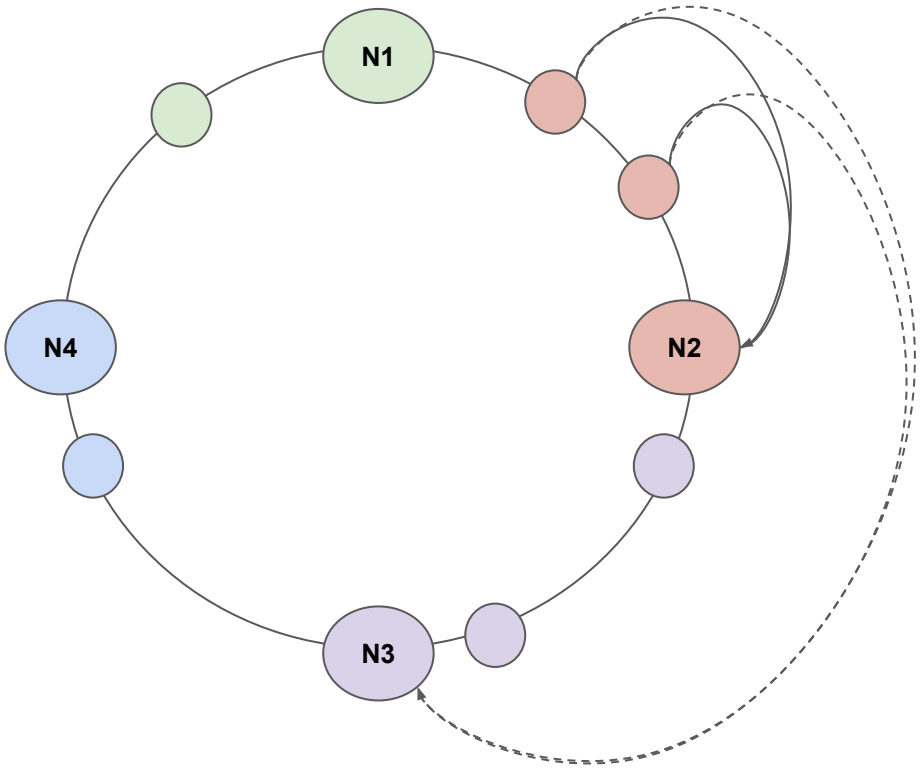
Le hachage cohérent nous permet donc d’ajouter ou de retirer des serveurs de l’ensemble sans avoir à rebalancer une grande partie des clés.

Mais qu’arrive-t-il si un serveur tombe?

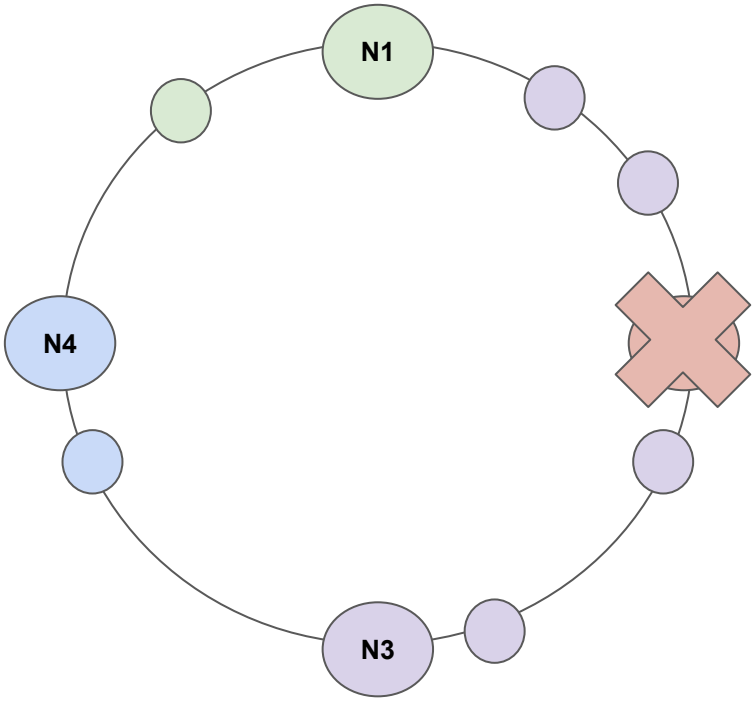
Encore une fois, la solution consiste à ajouter de la redondance. On pourrait avoir deux serveurs par plage pour s'assurer que si un serveur tombe en panne, sa plage sera tout de même disponible.

Comme l’algorithme de hachage cohérent prévoit déjà quel serveur doit servir les clés d’une plage lorsqu’on retire un serveur, une autre solution consiste à faire en sorte que les données d’une plage soient répliquées vers le ou les serveurs de la ou des plages suivantes. De cette manière, si un serveur tombe en panne, sa plage sera alors servie par son successeur qui possède déjà les données.

Réplication



Réplication vers
le nœud suivant.



N3 possède déjà
les données de
la plage de N2.

Noeuds virtuels

Au lieu d'associer une seule plage par nœud, plusieurs plages sont associées à chacun des nœuds ce qui rend l'ajout ou la suppression plus granulaire.