

# Autour de la distribution

Polytech Marseille

Simon Vilmin

[simon.vilmin@univ-amu.fr](mailto:simon.vilmin@univ-amu.fr)

2024 - 2025



# Plan

## Systèmes distribués

### Autour du théorème CAP

- cohérence, disponibilité

- Latence

### Réplication

- Primaire-secondaire

- Multi-primaire (multi-leader)

- Multi-noeuds

- Résumé

### Partition

- Par intervalle

- Par hachage de clé

- Rééquilibrage

- Routage des requêtes

- Réplication + partition

- Résumé



**Remarque :** la distribution des données n'est pas l'apanage des BD noSQL, le relationnel se distribue aussi

# Sources

Sources principales :

- *Designing Data-Intensive Applications*,  
Martin Kleppmann, O'reilly, édition 2021, [PDF](#) [Github](#)
- *Bases de données documentaires et distribuées*,  
Philippe Rigaux, version 2023, [cours en ligne](#)
- *Les bases de données NoSQL*,  
Rudy Bruchez, Eyrolles, édition 2015, [boutique](#)

## Dans l'épisode précédent

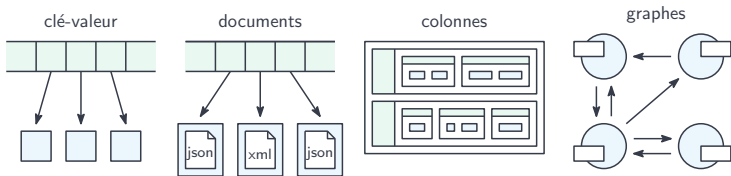
SGBDR très utilisés, encore aujourd'hui

- schémas de données très contraints
- favorisant l'optimisation de toutes les opérations

Mais, nouveaux besoins dans les années 2000-2010 :

- gestion de gigantesques volumes de données
- besoin de flexibilité et de disponibilité

Émergence des solutions *noSQL* (not only SQL).



## 3 principes

**?** **Question :** Qu'est-ce qu'on attend d'un système qui gère des données ?

3 objectifs :

- *fiabilité* : résistance aux erreurs logicielles, matérielles, humaines
- *maintenabilité* : facilité d'entretien, de mise à jour, compréhensible, ...
- *scalabilité* : capacité à *conserver les performances* face à une *augmentation de la charge*

**i** **Remarque :** scalabilité (autre formulation) = les performances du système sont *proportionnelles* aux *ressources* qui lui sont allouées (autre formulation)

# Scalabilité : quoi ?

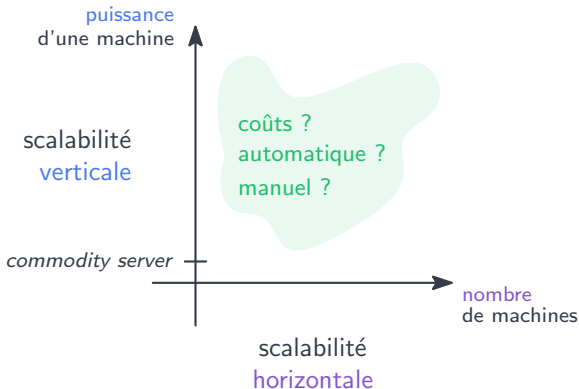
## ? Question : *charge*, *performance*, *ressource* ?

- *charge* : nombre de requêtes par secondes, ratio lecture/écriture, nombre d'utilisateur.ices parallèles, ...
- *performance* :
  - *débit*, quantité de données accessible par seconde (# de tuples, documents, ...)
  - *temps de réponse*, temps mis pour envoyer une requête et recevoir la réponse
- *ressources* : temps de calcul CPU, mémoire RAM et disque allouée, ...

## i Remarque :

- dépend du *type d'application* bien sûr, pour nous les *SGBD*
- *performance* estimée avec des *statistiques* (médiane, quantiles, ...)

# Scalabilité : comment ?



- scalabilité **verticale** (*scaling up*) : augmenter la **puissance** d'une machine/serveur
- scalabilité **horizontale** (*scaling out, shared-nothing*) : augmenter le **nombre** de machines/serveurs
- le choix n'est **pas binaire** !

**i Remarque :** les systèmes noSQL jouent sur la *scalabilité horizontale* et deviennent des *systèmes distribués*.

Système distribué :

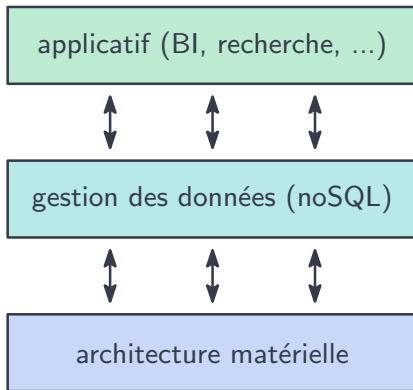
- système logiciel reposant sur un *réseau de machines/serveurs interconnectés*
- vise à coordonner ces machines pour mener à bien *une opération commune*
- les machines communiquent via *l'échange de messages*

Pour les systèmes noSQL :

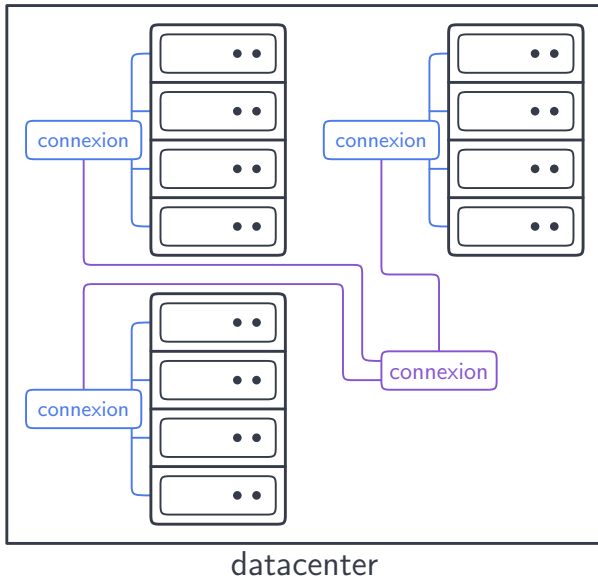
- cas particulier de système distribué
- objectif : gérer des *grandes masses de données* (distribuées du coup) !



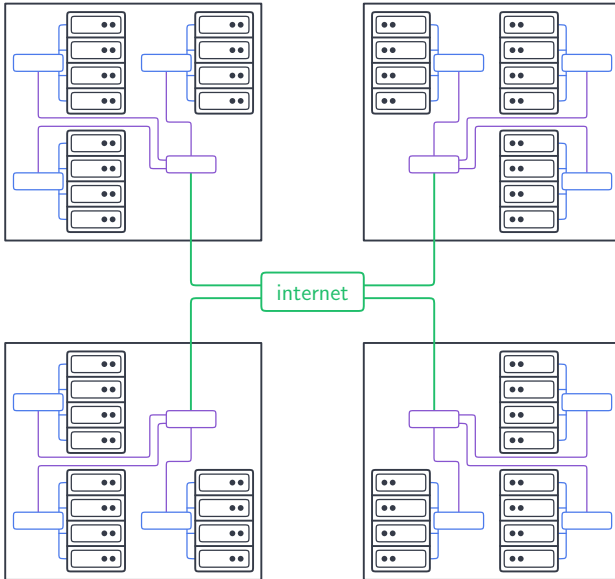
## Du stockage à l'utilisation



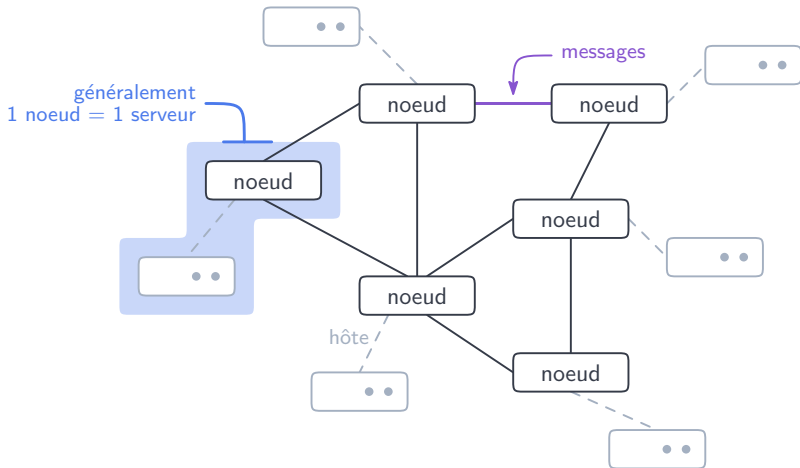
## Architecture matérielle : datacenter



# Architecture matérielle : grille



# Réseau distribué noSQL



**Astuce :** en bref, ensemble de *noeuds* (processus) échangeant des *messages* (réseau).

# Objectif

**?** **Question** : on a un système distribué, mais *comment on y répartit les données* ?

Deux idées complémentaires

- *réplications* : *copier* les données sur plusieurs noeuds
- *partition* (sharding) : *partitionner* et répartir les données sur les noeuds

Pour favoriser

- la *scalabilité*
- la *disponibilité* / la *tolérance aux pannes* : si une machine tombe en panne, une autre prend le relais
- le *temps de réponse* : une requête envoyée à une machine géographiquement proche mettra moins longtemps à voyager

# Autour du théorème CAP

Systèmes distribués

Autour du théorème CAP  
cohérence, disponibilité  
Latence

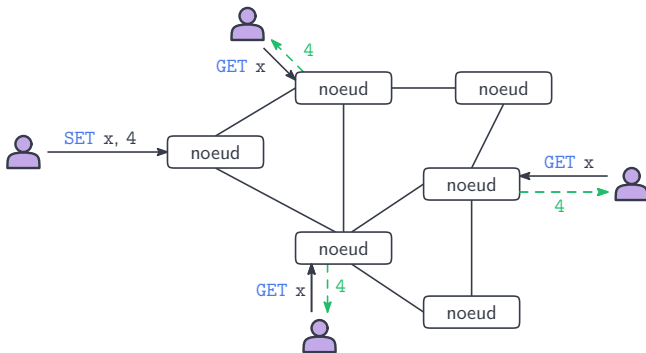
Réplication

Partition

**i Remarque : *ENCORE?*** Jipèp. Oui, mais les notions de *cohérence* et de *disponibilité* vont nous être utiles. Et ça permet de discuter du modèle *PACELC*.

## cohérence

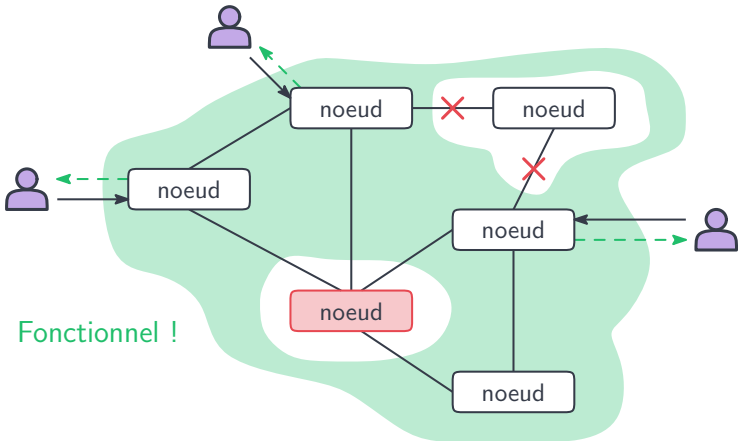
*cohérence* : peu importe le noeud d'accès, les données que l'on voit sont les mêmes



**! Attention :** à ne pas confondre avec la cohérence de **ACID** qui se réfère à la *logique des données*

# Disponibilité

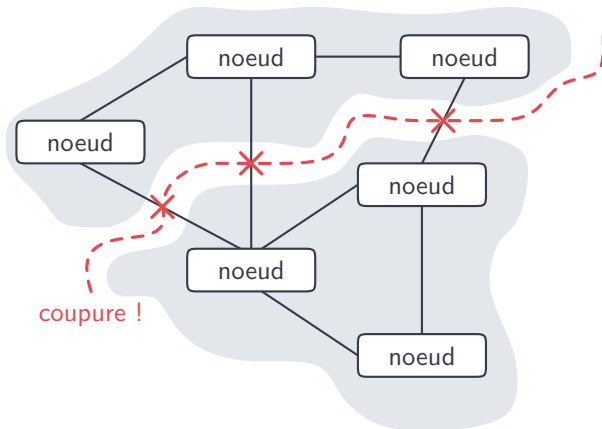
*Disponibilité* : le système doit être opérationnel en cas de pannes de noeuds, de *partitionnement du réseau*, etc ...





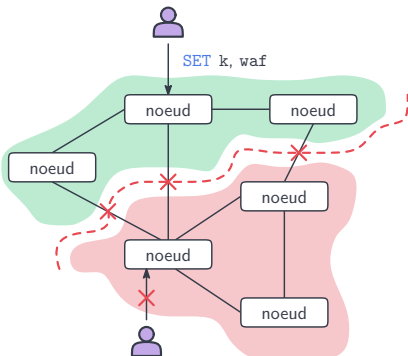
# Partitionnement

*Partition* : dû à un problème réseau (par ex), le système est coupé en (au moins) deux parties contenant plusieurs noeuds chacune.

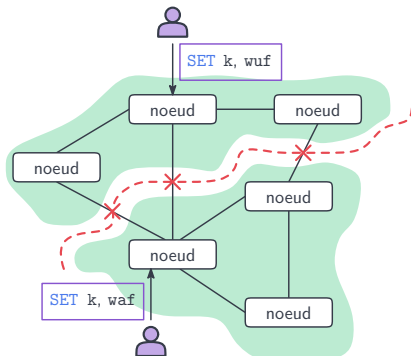


# Théorème CAP : les faits

Une *partition* est un *impondérable*. Quand ça arrive, on ne peut pas garantir ET *cohérence* ET *disponibilité*.



Consistence



Disponibilité

# Théorème CAP : l'histoire

## **i** Remarque :

- principe énoncé en 2000 Éric Brewer
- a amené une réflexion sur les systèmes de gestions de données distribuées !
- « preuve » dans un contexte *très restreint* en 2002 par Gilbert et Lynch

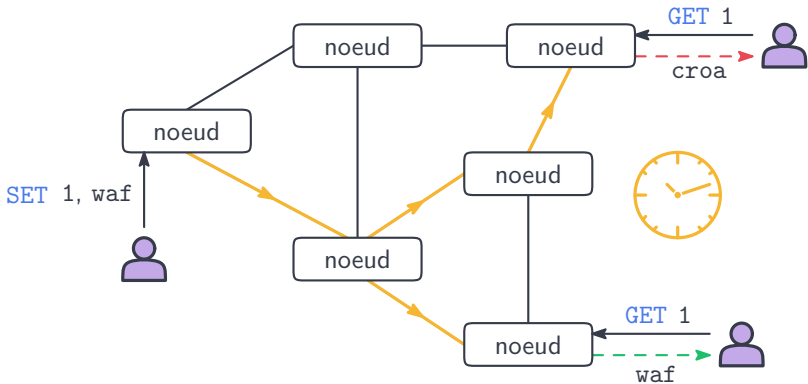
## **×** Problème :

- a mené à des classifications type AC, AP, CP *trompeuses*
- en l'absence de partition, on peut très bien avoir A et C !
- quid des autres paramètres possibles ?

**?** Question : en l'absence de problème réseau, notre système est donc parfait ? *NON*

## cohérence vs Latence

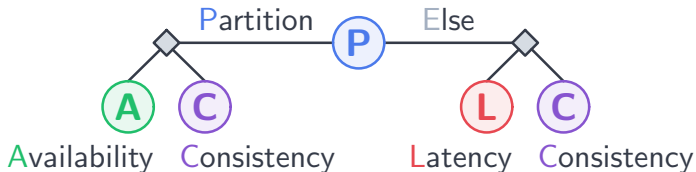
Après une modif, le système entier doit être mis au courant. Pendant cette période de *latence*, des *incohérences sont possibles* !



# PACELC

**? Question :** que faire? Bloquer temporairement l'accès pour avoir la *cohérence*? Ou autoriser une incohérence pour une *latence* minimum et donc une meilleure *disponibilité*?

**i Remarque :** c'est au coeur du modèle *PACELC* de Daniel Abadi.




# Vraiment mieux que CAP ?

## Remarque :

- PACELC *outil pratique* pour s'orienter
- mais *forcément incomplet* car très simple

## Question : quid de la gestion des *conflits* et de la *concurrence* ?

 **Rappel :** il y a un *équilibre à trouver* entre tous ces aspects, qui dépendra des cas !

## Astuce :

- ressources sur [AMeTICE](#) concernant CAP, PACELC, ...
- plus de résultats du côté de la *théorie des systèmes distribués* !

## cohérence à terme

### Remarque :

- « choix » à faire entre la *latence* et la *cohérence immédiate*
- mais cohérence « au bout d'un certain moment » !

*cohérence à terme* qu'on avait évoqué dans les propriétés *BASE* de moult moteurs *noSQL* : la base de données *sera* consistante à un moment.

Plusieurs niveaux de cohérence :

1. *cohérence faible* : aucune garantie sur la cohérence des données, même à termes (peu populaire)
2. *cohérence à terme* : les données seront consistantes *à un moment dans la vie de la BD* (populaire en noSQL)
3. *cohérence forte* (cohérence, linearizability, atomic consistency, ...) : quel que soit le noeud, la donnée à laquelle on accède est la même

# Réplication

## Systèmes distribués

## Autour du théorème CAP

### Réplication

- Primaire-secondaire

- Multi-primaire (multi-leader)

- Multi-noeuds


- Résumé

### Partition


**i Remarque :** en bref, on brosse le portrait de *3 stratégies de réplication*. Chacune va nous permettre de mettre le doigt sur des *difficultés liées à la distribution*.



# Définition

 **Définition** : la *réplication* est la copie des mêmes données sur plusieurs machines connectées via un réseau. Un noeud qui contient une copie des données est un *réplicat*.

 **Remarque** : rejoint le principe de *redondance*

 **Question** : comment faire pour que *chaque réplicat* soit à jour sur les données ?

Plusieurs stratégies possibles :

- *primaire / secondaire* (leader/follower, actif/passif, maître/esclave)
- *multi-primaires* (multi-leader)
- *multi-noeuds* (leaderless)

# Primaire-Secondaire : principe

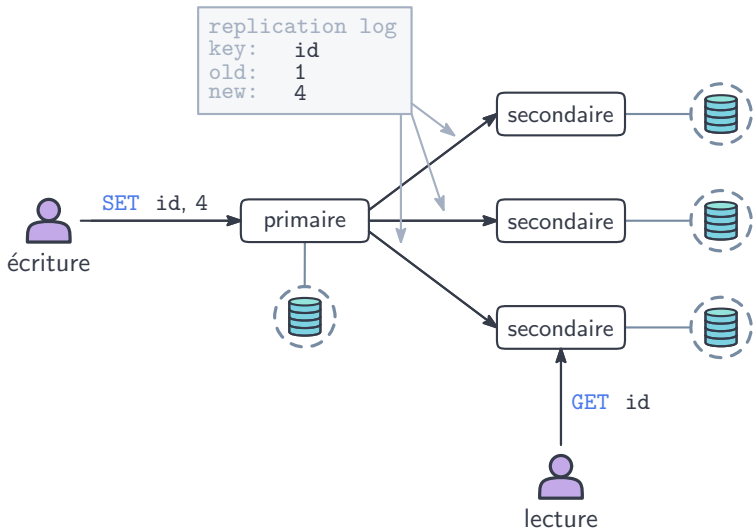
Structure :

1. un noeud *primaire*
2. les autres sont *secondaires*

Fonctionnement lecture / écriture :

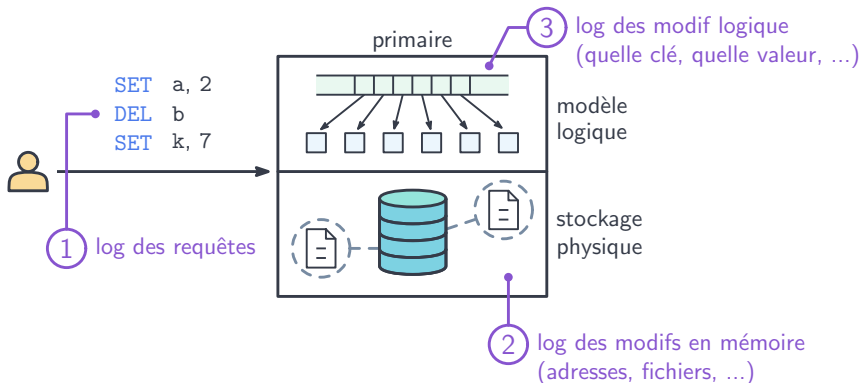
1. requêtes d'*écriture* faites au *primaire*, qui écrit d'abord la donnée en local et l'envoie aux *secondaires* sous forme de *replication log*
2. en cas d'*écriture*, les *secondaires* reçoivent le *replication log* du primaire pour faire la mise à jour en local
3. les requêtes de *lectures* faites sur le *primaire* ou les *secondaires*

# Schéma



# Replication log

**?** Question : à quoi ressemble ce *replication log* ?



## En texte

1. log des *requêtes* ( $\simeq$  forward aux secondaires) :
  - problème du *déterminisme* : que se passe-t-il quand la requête fait appel à de l'aléatoire ? différentes valeurs sur chaque noeud ?
  - attention à l'ordre des requêtes de modification !
2. log des modifications en *mémoire* :
  - *Write-Ahead Logging (WAL)* : journal des modifications faites en mémoire
  - *dépendant du matériel* et des versions de logiciels de chaque machine
3. log des modifications au *niveau logique* :
  - description des modifications, suppressions, mises à jour de documents/tuples, ...

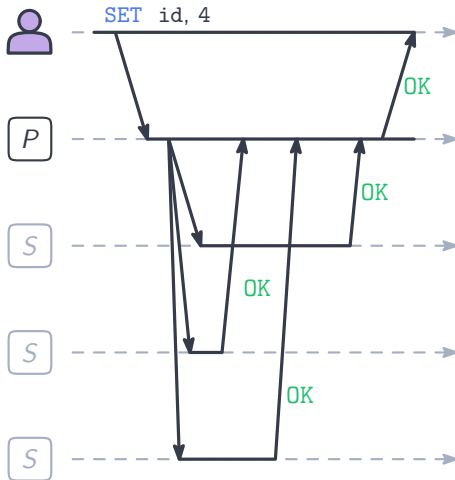
**?** **Question** : le *primaire* est censé *valider* la requête d'écriture, mais *quand le fait-il*?

**✓** **Réponse** : trois choix

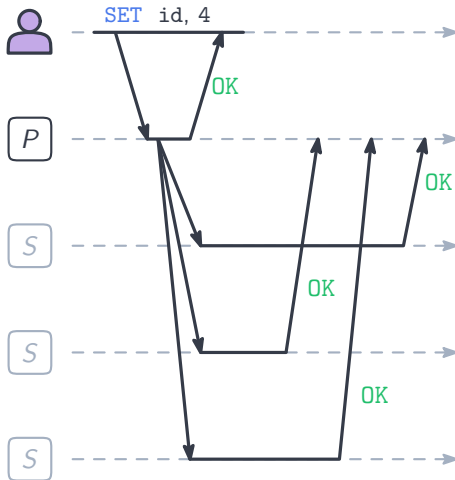
1. *synchrone* : il attend la validation de *tous les secondaires*
2. *asynchrone* : dès qu'il a fait la modif en local, sans attendre les *secondaires*
3. *un mélange des deux* : par ex *un secondaire synchrone* par sureté, et les autres *asynchrones*

- le synchrone garantit une copie valide des données
- mais si le noeud est en panne, la requête ne peut aboutir !
- en plus de ralentir l'écriture en général

## Exemple synchrone



## Exemple asynchrone





# Replication lag


l'asynchrone peut entraîner un *replication lag* : on essaye de lire une donnée avant qu'elle ne soit modifiée partout


Quelques exemples :

- lire des données que l'on vient d'écrire (*read your own writes*)
- accéder plusieurs fois à la même donnée, mais sur des secondaires différents pas forcément à jour (*monotonic reads*)

**i Remarque :** ces exemples forment des *niveaux intermédiaires* de *cohérence* !

## Noeud en panne

 **Question** : Que se passe-t-il si un noeud tombe en panne ?

 **Réponse** : si le noeud est *secondaire*, pas de problèmes

Au redémarrage du noeud

- il a stocké en mémoire les opérations effectuées avant l'arrêt (log)
- demande au primaire les opérations à partir desquelles reprendre les modifications

## Primaire en panne

**?** **Question :** comment faire si le *primaire* s'arrête ?

1. il faut *déterminer qu'il est off* : par exemple avec des ping réguliers entre les noeuds et un timeout
2. *choix d'un nouveau primaire* : algorithme de vote à la majorité type **Paxos**, ou choisi par un noeud « contrôleur » (aussi élu).
3. *reconfigurer le système avec le nouveau primaire* : re-router les requêtes vers le nouveau primaire

**i** **Remarque :** En général, le meilleur candidat pour devenir *primaire* est celui qui a les données les *plus à jour*

## Multi-primaire (multi-leader) : principe

**i Remarque :** dans un primaire-secondaire, *les requêtes d'écriture* passent par le *primaire* ... si *problème de connexion avec le primaire*, plus *aucune écriture possible*

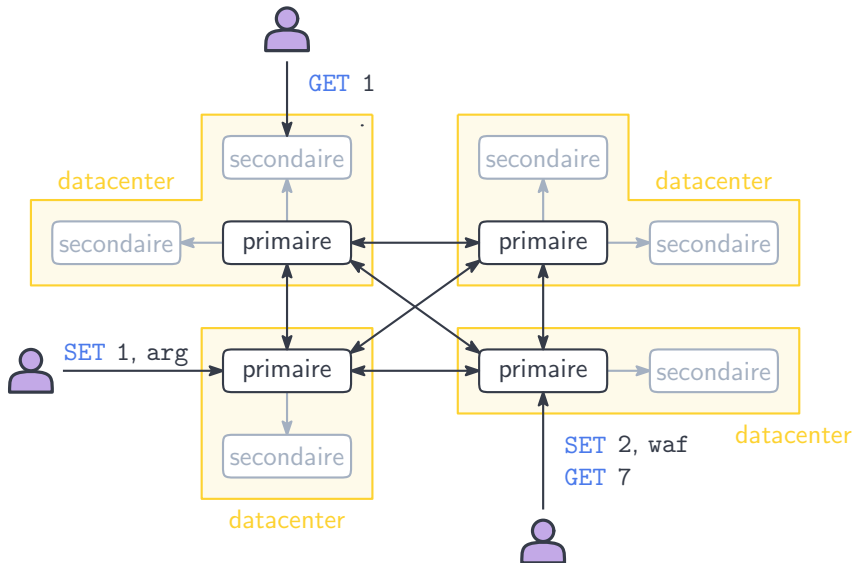
**💡 Idée :** mettre *plusieurs primaires* ! C'est l'architecture *multi-primaire*

Fonctionnement *ressemblant au primaire-secondaire* :

- un primaire transfère les changements à *tous les autres noeuds*
- un primaire agit comme un secondaire pour les autres primaires
- généralement *asynchrone*

**i Remarque :** pour transférer des changements, possible de passer par des *primaires intermédiaires* !

## Multi-primaire : schéma



## Pourquoi c'est intéressant ?

Applications/services avec plein de datacenter :

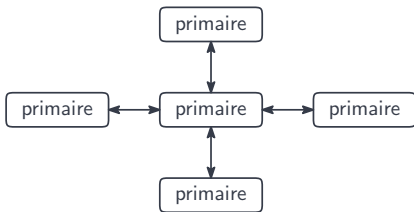
- 1 primaire par datacenter, chaque datacenter est en primaire-secondaire
- chaque datacenter est indépendant des autres

Applications avec *opérations hors-ligne* :

- un client contient un primaire, qui garde les modifications en local
- au retour de la connexion, synchronisation avec tous les autres clients
- ex : calendrier, mails, ...

 **Remarque** : similaire aux *outils collaboratifs* type Google Doc !

# Topologies

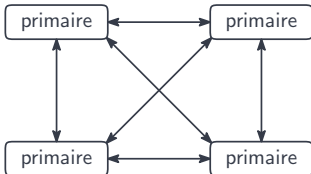
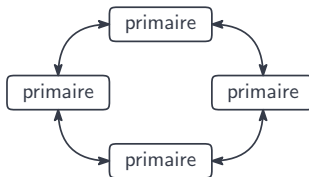


## étoile

- passage par le primaire central
- en cas de panne, **isole les autres** (en attente d'une autre étoile)

## anneau / circulaire

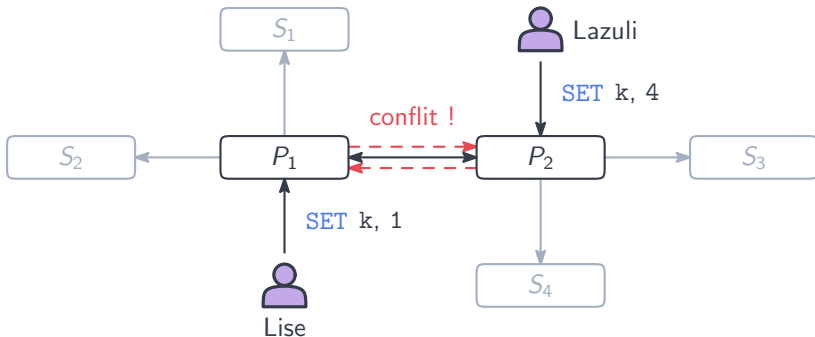
- sensiblement identique à étoile



## connexion complète

- la **plus courante**
- moins de problèmes de déconnexion
- mais risques dans l'ordre de réception des messages

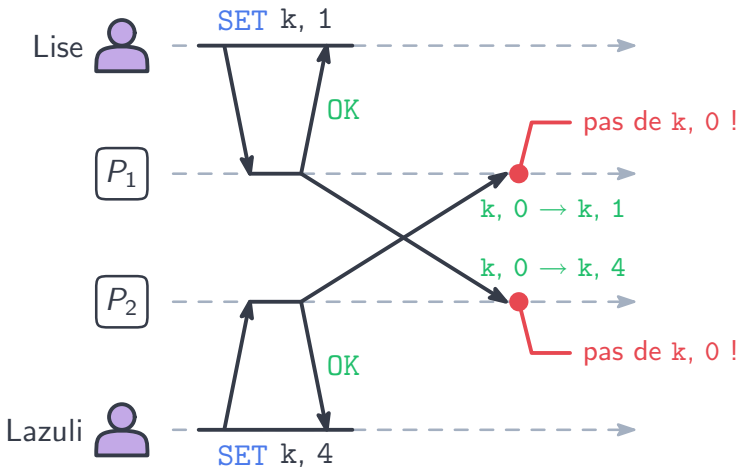
Il y a un os



❌ **Problème :** en cas d'*écritures simultanées de mêmes données sur des primaires différents*, la réplication des données va rencontrer un *conflit*



## Le même problème, version temporelle



## Gestion des conflits en une slide

*les éviter!* par ex en disant qu'une valeur n'est modifiée que par un primaire

- « les clés  $k$ ,  $m$ , sont modifiées via  $P_1$  »
- « les clés pouet,  $i$ , sont modifiées via  $P_2$  »

*consensus sur les valeurs*

- timestamp sur les requêtes pour savoir qui est le premier
- fusionner les valeurs (dans notre exemple,  $k$ , 4/1)

*faire remonter le conflit* à l'application cliente et sa logique de résolution

- dès l'écriture, au moment du conflit
- à la prochaine lecture des données (les conflits sont stockés et remontés à ce moment-là)



**Remarque :** problème sujet à des recherches !

## Multi-noeuds : principe

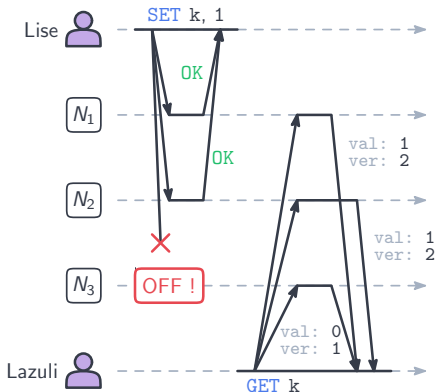
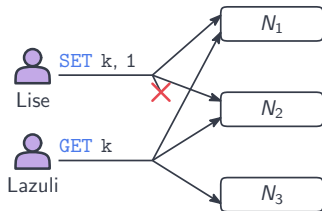
**? Question :** pourquoi forcément mettre des *primaires/secondaires*?

Architecture *multi-noeuds* :

- chaque noeud reçoit *lectures et écritures*
- une requête est transmise à *tous les noeuds*
- notion de *quorum*, sur  $n$  noeuds :
  - chaque *écriture* doit être validée par  $w$  noeuds
  - chaque *lecture* est faite sur  $r$  noeuds

**i Remarque :** en pratique, on utilise aussi de la *partition*. Une donnée est sur *au plus  $k < n$  noeuds*  $\implies$  la requête n'est envoyée qu'aux noeuds avec la donnée

## Multi-noeuds : schéma



**i Remarque :** on suppose  $n = 3$ ,  $w = 2$ ,  $r = 2$  et déjà une paire  $(k, 0)$

## Que s'est-il passé là ?

Paramètres :

- $n = 3$  noeuds,  $w = 2$  écritures requises,  $r = 2$  lectures requises
- on a une clé  $k$  avec sa valeur (`var`) 0, version (`ver`) 1

Lise veut remplacer  $(k, 0)$  par  $(k, 1)$  :

- mise à jour **OK** pour  $N_1, N_2$
- **non** pour  $N_3$  qui a un problème
- opération **validée** car  $w = 2$  !

Lazuli veut récupérer la valeur associée à  $k$  :

- $N_1, N_2$  **renvoient bien la nouvelle valeur**, version 2
- $N_3$  est revenu mais ***pas à jour ! sa version est plus ancienne***  $\implies$  la valeur n'est pas gardée

## Read repair et anti-entropy

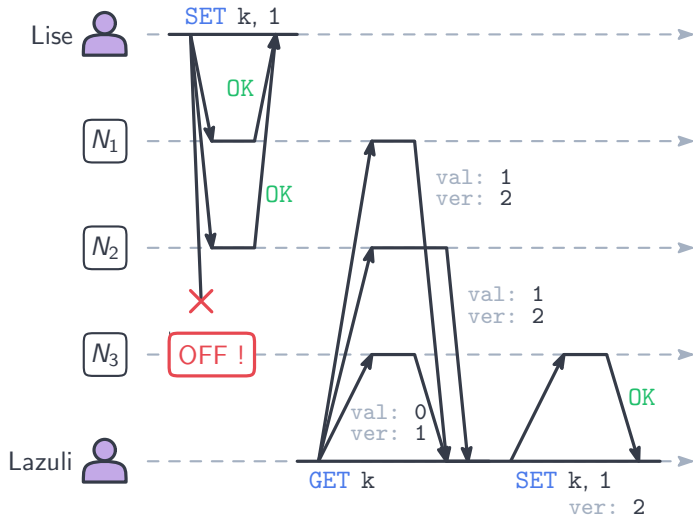
**i Remarque :** *génial pour la disponibilité* ça ! quand il y a un noeud en panne, c'est presque transparent, mais par contre ...

**? Question :** Plus de primaire pour dire aux autres quoi changer et dans quel ordre (replication log) ... comment garantir la *cohérence à termes* ?


Deux moyens :


- *read repair* : on corrige une valeur « périmée » au moment de *la lire*
- *anti-entropy* : un processus en tâche de fond qui copie (un beau jour) les données manquantes

## Read repair sur notre exemple



## Choix des valeurs du quorum

 **Question :** comment choisir  $r$  et  $w$  de sorte qu'on accède toujours à *au moins 1 valeur correcte* ?

 **Réponse :** la cohérence est *assurée* si  $r + w > n$

Le choix de  $r$  et  $w$  devient une balance entre *cohérence* et *latence*

- $r$  et  $w$  grands veut dire *données sûres* mais *beaucoup de latence*
- possible de favoriser  $r$  ou  $w$  en fonction des besoins
- choix courant :  $n$  impair,  $w = r = \lfloor n/2 \rfloor + 1$



# Conflits

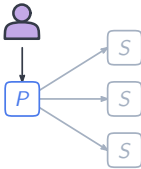
**i Remarque :** multi-noeuds *tolérants aux pannes* et *intéressants pour la cohérence* ... mais on a à nouveau le souci des *conflits*!

Mêmes stratégies que pour les multi-primaires :

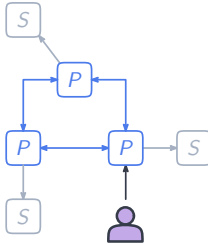
- essayer de les éviter
- consensus sur les valeurs
- faire remonter le conflit

**i Remarque :** voir le *livre* de Martin Kleppmann pour plus de détails!

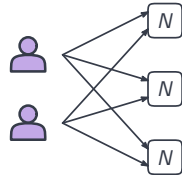
# Résumé




primaire-secondaire



multi-primaires



multi-noeuds

 **Rappel** : 3 architectures, avec des propriétés différentes :

- *primaire-secondaire* : populaire (car simple), sans conflits, mais peu tolérant aux pannes
- *multi-primaire* : plus robuste, mais conflits, généralise primaire-secondaire
- *multi-noeuds* : plus robuste, quorum pour cohérence/latence, mais conflits

# Partition

Systèmes distribués

Autour du théorème CAP

Réplication

Partition

- Par intervalle

- Par hachage de clé

- Rééquilibrage


- Routage des requêtes

- Réplication + partition

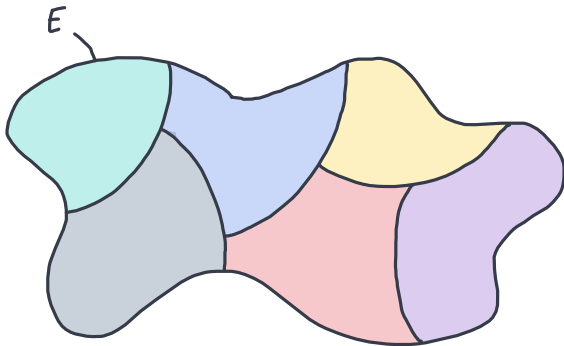
- Résumé

**i Remarque :** en bref, il y a *deux manières* de partitionner les données : par *hach*, par *intervalle*. La partition se combine avec la réplication.

## Parti-quoi ?

 **Définition** : Soit  $E$  un ensemble. Une *partition* de  $E$  est une famille  $\mathcal{F} = \{F_1, \dots, F_m\}$  de sous-ensembles (non-vides) de  $E$  telle que :

- $F_i \cap F_j = \emptyset$  pour tout  $F_i, F_j \in \mathcal{F}$ ,  $F_i \neq F_j$
- $\bigcup_{i=1}^m F_i = E$



## Pour nos données

**? Question :** Pourquoi partitionner nos données sur plusieurs noeuds ?

**✓ Réponse :** pour répartir un *immense jeu de données* sur plusieurs machines

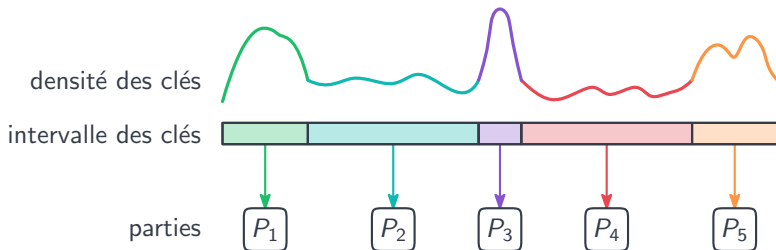
**i Remarque :**

- on appelle parfois le partitionnement du *sharding*
- on aimerait répartir les données *équitablement*
- on va regarder sur les données de type (clé, valeur)
- on parle juste de partition ici, mais c'est généralement utilisé conjointement avec la *réplication*

# Partitionnement par intervalles

**i Remarque :** on suppose que les clés sont *ordonnées*

1. partitionner l'ensemble (ordonné) des clés par *intervalles de valeurs* :
  - les clés sont des dates → intervalle de temps
  - entrées d'une encyclopédie → intervalle de termes
  - nombre entiers → intervalles classiques
2. un noeud peut prendre en charge plusieurs parties de la partition



## Quelques propriétés

**i Remarque :** le partitionnement par intervalles est « sémantique », il *reflète un peu la structure des données*

Conséquences :

✓ efficace pour chercher des *intervalles de clés* (*range queries*)

✗ mais très dépendant de la *distribution* des clés,

→ introduit un *déséquilibre dans les parties* et donc un déséquilibre de charge sur le système !

# Partition par hachage de clé

*fonction de hachage :*

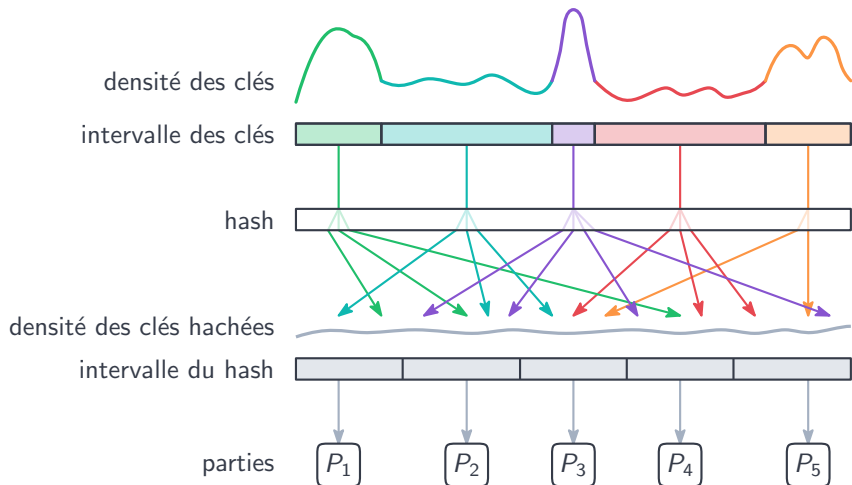
- calcule un entier (souvent entre 0 et  $2^{32} - 1$  ou  $2^{64} - 1$ ) à partir d'une clé
- propriétés souhaitées :
  - fonction *injective*
  - distribue les hash de *manière uniforme* sur l'intervalle

Principe : *partition par hachage*

1. utilisation de la fonction de hachage sur les clés
2. partition de l'intervalle des valeurs de hachage
3. répartition sur les noeuds





# Schéma



## Quelques propriétés

**i Remarque :** la fonction de hachage distribue uniformément les valeurs dans l'intervalle, on peut *séparer les données par des intervalles « uniformes »*

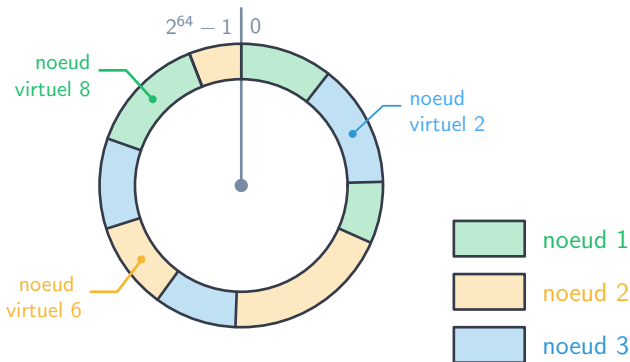
Conséquences :

-  plus possible de chercher des intervalles de clés
-  plus facile de séparer les données, notamment avec des intervalles de tailles similaires

# Anneau, noeuds virtuels

## **i** Remarque :

- parfois l'espace des clés ou des hash est vu comme un anneau : *la dernière valeur boucle sur la première*
- la partition *logique* ne correspond pas forcément à la partition *physique*. Les *parties de la partition* des fois appelés *noeuds virtuels*.



# Rééquilibrer les partitions

**i Remarque :** les partitions peuvent se déséquilibrer, certains noeuds vont être plus sollicités que d'autres, ... besoin de *rééquilibrage*!

**! Attention :** pour le rééquilibrage, il faut *minimiser le nombre de données à déplacer* (si jamais)

## 3 idées

- *partition dynamique* : on coupe les parties qui deviennent trop grosses
- *nombre de parties fixé* : on fixe à l'avance un grand nombre de partitions plus petites
- *hachage cohérent* : chaque noeud introduit  $m$  noeuds virtuels répartis par hachage sur l'anneau

# Rééquilibrage dynamique

quand une partie *excède une certaine taille*, elle est *divisée en deux parties* de tailles  $\simeq$  égales (*split*), via la médiane par exemple.

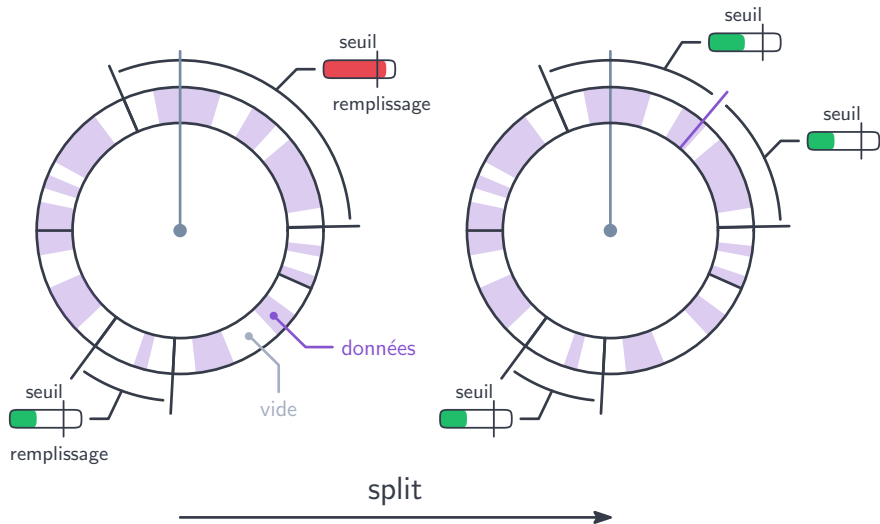
Valable pour :

- partition par intervalles
- partition par hachage

Propriétés :

- taille d'une partie constante, fixée par un seuil
- nombre de parties augmente avec la taille des données
- les nouvelles parties sont stockées sur les noeuds moins chargés

# Schéma



## Nombre fixé de parties

Pour  $n$  noeuds

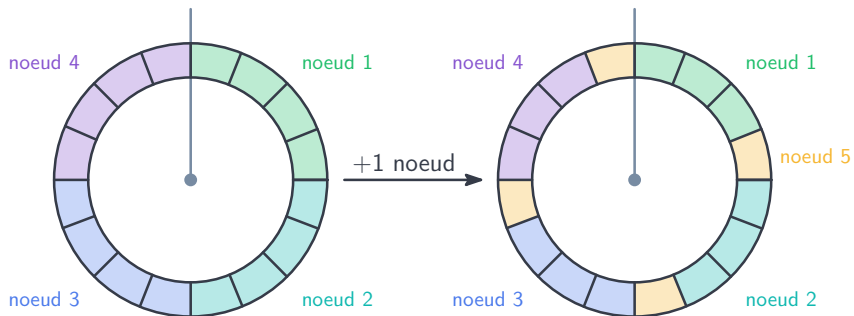
- on divise l'espace en  $m \gg n$  parties (noeuds virtuels) de même taille
- chaque noeud stocke  $m/n$  parties
- un nouveau noeud « *chipe* » *le même nombre de parties* à tous les autres

Utilisable pour *hachage par clé* (car répartition uniforme des données)

Propriétés :

- taille d'une partie augmente avec la taille des données
- nombre de parties constant
- entre les noeuds, on ne bouge que des parties d'un seul bloc

# Schéma





## Hachage cohérent

- chaque noeud va avoir  $m$  *noeuds virtuels* placé sur l'anneau avec une *fonction de hash*
- une clé appartient au premier noeud rencontré *dans le sens horaire*
- l'ajout d'un noeud rajoute  $m$  intervalles

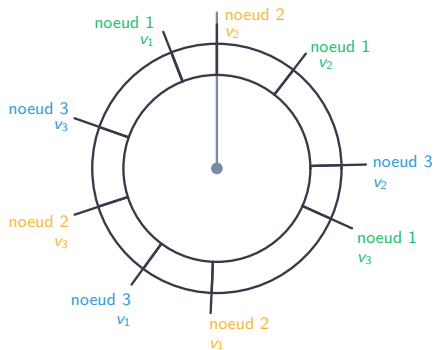
Utilisable pour *hachage par clé* (toujours l'uniformité)

Propriétés :

- nombre de parties proportionnel au nombre de noeuds
- taille des partitions augmente avec la taille des données ...
- mais réduit avec le nombre de noeuds !

**i Remarque :** avoir  $m$  noeuds virtuels par noeud permet de tendre vers l'*homogénéité de la charge*

# Schéma



$m = 3$



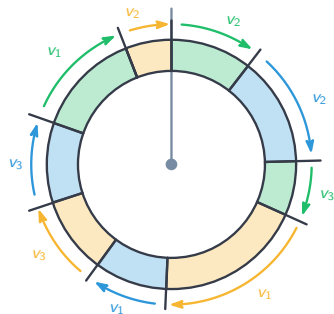
noeud 1




noeud 2




noeud 3



## Où vais-je ?

 **Question** : comment savoir sur *quelle partition* se trouve une donnée ?

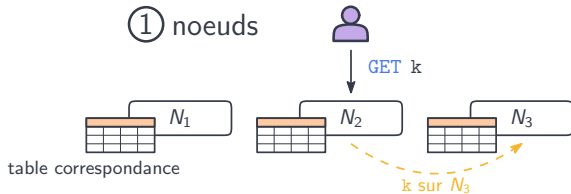
 **Réponse** : au moyen d'une *table de correspondance* qui fait le lien entre *intervalle de partition* et *noeud*

3 possibilités de stockage :

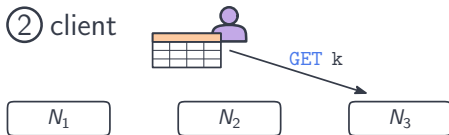
1. sur *chacun des noeuds*
2. directement sur le *client*
3. sur un *noeud tiers* qui sert de *routeur*

# En dessin

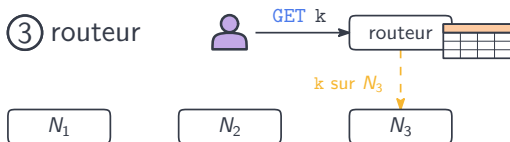
① noeuds



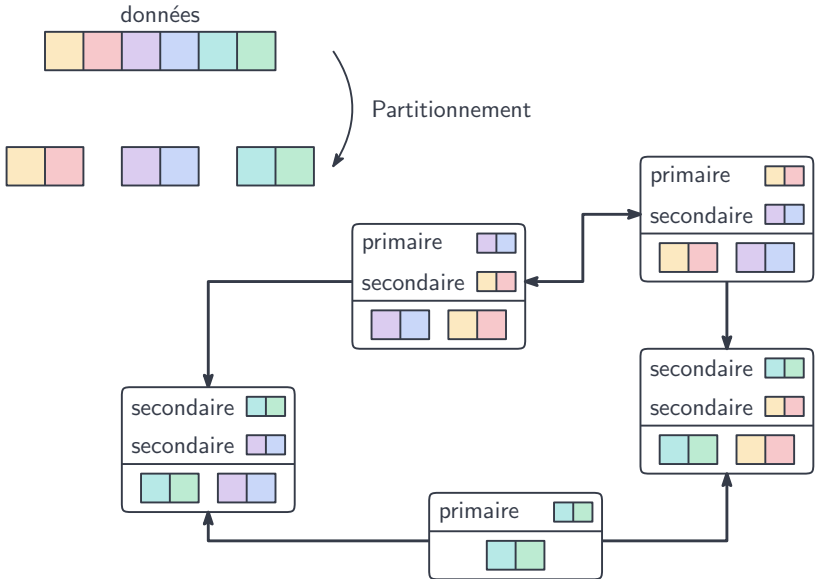
② client




③ routeur




## Exemple de partition + réplication




# Résumé


 **Définition** : *partition (sharding)* : *séparation* et *répartition* des données sur divers noeuds. Utilisé conjointement avec la *réplication*.

 **Rappel** : Deux techniques de partitionnement

- par *intervalle* : on partitionne *directement sur les clés*
- par *hachage* : les clés sont *hachées puis partitionnées*

 **Rappel** : Pour rééquilibrer des partitions entre les noeuds

- partition *dynamique* : si une partie est trop remplie, on la coupe en 2
- nombre de parties *fixé* : le nombre de parties est fixé à l'avance
- *hachage cohérent* : un noeud stocke *m* parties hachées sur l'anneau

 **Remarque** : Pour diriger les requêtes : *table de correspondance* stockée sur le *client*, les *noeuds* ou un *routeur tier*