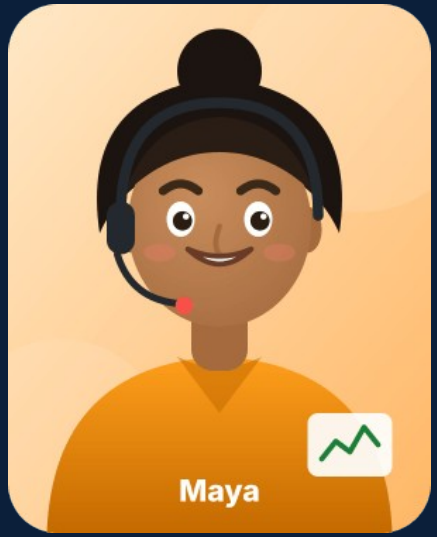
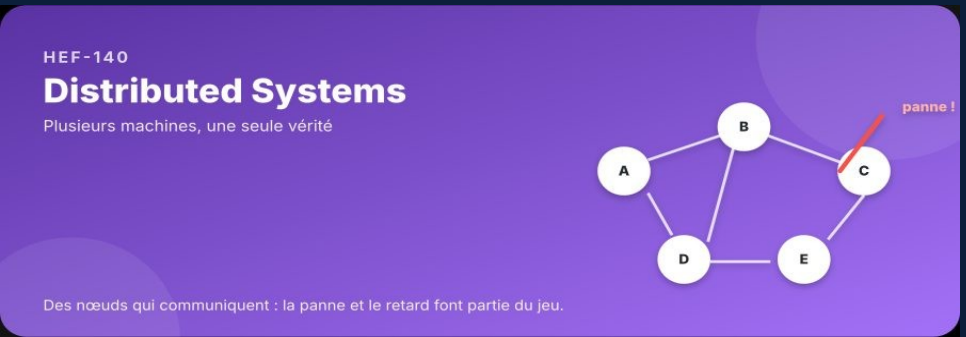
HEF-140 · MISSION 06 / 10

Partitionnement



Quand les données ne tiennent plus sur une machine, on les découpe. Le choix du découpage décide de tout : équilibre, requêtes possibles, coût des changements.

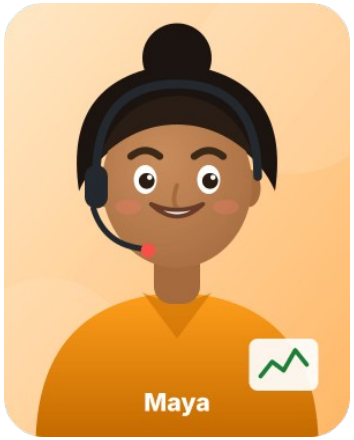


Badge : Découpeur de données

100 min · avancé

L'histoire

La table des inscriptions de HashBox atteint 2 To. Aucune machine abordable ne la contient confortablement, et les écritures saturent le disque. L'équipe décide de répartir les données sur 5 serveurs : « l'inscription numéro `n` va sur le serveur `n mod 5` ». Trois mois plus tard, il faut passer à 6 serveurs. Verdict : **83 % des données doivent changer de serveur**. Le déménagement dure quatre jours, pendant lesquels le service est dégradé.



> *« Le découpage par modulo est le plus simple et le pire quand on grandit, dit Sam. Il existe une technique, le hachage cohérent, où ajouter un serveur ne déplace que 1/6 des données. Et il y a d'autres pièges : les clés populaires qui surchargent un serveur, les requêtes qui doivent interroger tous les serveurs. »*

L'histoire (suite)

Tu apprends à choisir une clé de partitionnement, à comparer hachage et plages, à implémenter un anneau de hachage cohérent avec nœuds virtuels, à mesurer le déséquilibre et à traiter les points chauds.

Ta mission

À la fin, je serai capable de…

1. expliquer pourquoi et comment on partitionne (sharding)
2. comparer partitionnement par hachage et par plage (requêtes, équilibre)
3. implémenter un hachage cohérent avec nœuds virtuels et mesurer la quantité de données déplacées
4. mesurer l'équilibre de charge et traiter un point chaud
5. identifier les requêtes inter-partitions et leur coût

Défi d'échauffement

Une bibliothèque de 1 million de livres est trop grande pour un seul bâtiment. Option A : un bâtiment par lettre initiale du titre. Option B : on attribue chaque livre à un bâtiment par un tirage « au hasard mais reproductible » à partir du titre. Pour chaque option, comment cherches-tu « tous les livres de A à C » ? Comment cherches-tu « le livre *Les Misérables* » ? Que se passe-t-il le jour où un nouveau bâtiment ouvre ? Quelle option évite qu'un bâtiment soit plein de titres commençant par « L' » ?

Le concept

Pourquoi partitionner

Raison	Explication
Volume	les données dépassent la capacité d'une machine
Débit	une seule machine ne suit pas les écritures ou les lectures
Latence	placer les données près des utilisateurs (par région)

Partitionner (*sharding*) = répartir les données en sous-ensembles, les **partitions** (*shards*), chacun sur une machine (souvent répliqué : mission 5). Ne pas confondre : **réplication** = copier les mêmes données ; **partitionnement** = **diviser** les données. On combine les deux.

Le concept

La clé de partitionnement

Chaque enregistrement possède une **clé** qui détermine sa partition. Le choix de la clé est **la** décision de conception : elle doit :

1. **répartir** uniformément données et trafic ;
2. permettre aux requêtes fréquentes de n'interroger qu'**une** partition ;
3. être **stable** (ne pas changer pendant la vie de l'enregistrement).

Deux grandes familles

	Par hachage	Par plage
Principe	`partition = hash(clé) mod N` (ou anneau)	chaque partition couvre un intervalle de clés ordonnées

Le concept

Deux grandes familles (suite)

	Par hachage	Par plage
Équilibre	**bon** (le hachage disperse)	dépend de la distribution des clés
Requête sur une clé	1 partition	1 partition
Requête sur une **plage**	toutes les partitions (*scatter-gather*)	1 ou quelques partitions
Point chaud typique	clé très populaire	clés **monotones** (horodatages : toutes les écritures vont à la dernière plage)

Le piège du `hash mod N`

Si `N` change, presque **tout** change de place :

Le concept

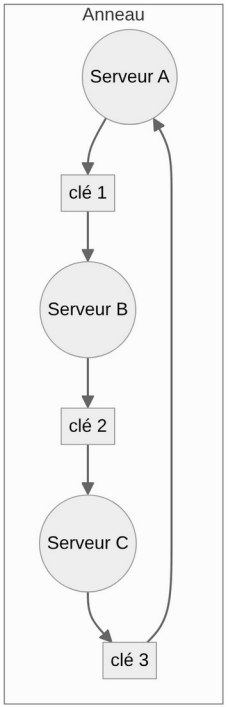
Le piège du `hash mod N` (suite)

```
N = 5 → 6 : une clé garde sa partition avec probabilité ≈ 1/6 ; ≈ 5/6 des clés se
   déplacent.
```

Le hachage cohérent

On place les **serveurs** et les **clés** sur un **anneau** (un cercle de valeurs de hachage). Une clé appartient au **premier serveur rencontré en tournant** dans le sens horaire.

Le hachage cohérent (suite)



Le concept

Le hachage cohérent (suite)

Description du schéma : un anneau où se succèdent les serveurs A, B, C et des clés ; chaque clé est affectée au serveur suivant dans le sens de rotation.

Ajouter un serveur ne déplace que les clés **situées juste avant lui** : en moyenne **1/(N+1)** des clés, au lieu de N/(N+1).

Nœuds virtuels : chaque serveur physique occupe **plusieurs** points de l'anneau (100 à 500). Ainsi la charge est répartie plus uniformément (loi des grands nombres) et un serveur qui tombe répartit sa charge sur **tous** les autres plutôt que sur un seul voisin.

Points chauds

Un hachage uniforme équilibre le *nombre* de clés, **pas** le *trafic* : une clé qui reçoit 30 % des requêtes (un cours en promotion) surcharge sa partition. Remèdes :

Le concept

Points chauds (suite)

Remède	Principe	Coût
Cache devant la partition	la lecture ne touche plus la partition	invalidation
Réplicas de lecture pour la clé	répartir les lectures	cohérence
Salage de la clé (`cle#0` … `cle#k-1`)	découper une clé chaude en k sous-clés	lectures par **scatter-gather** sur k sous-clés
Partition dédiée	isoler les clés chaudes connues	gestion

Requêtes inter-partitions

Requête	Coût
Lecture par clé	1 partition

Le concept

Requêtes inter-partitions (suite)

Requête	Coût
Jointure ou agrégat global	interroger **toutes** les partitions, puis fusionner : latence = la plus lente
Transaction touchant 2 partitions	coordination (2PC, saga : mission 9)
Index secondaire global	maintenu sur chaque partition (index local) ou séparé (index global, mise à jour asynchrone)

D'où la règle : **regrouper** ensemble ce qui est accédé ensemble (par exemple partitionner les inscriptions **par cours** si on les interroge par cours).

Rééquilibrage

Le concept

Rééquilibrage (suite)

Quand on ajoute des machines ou qu'une partition grossit : déplacer des partitions entières (toujours plus de partitions que de machines : 10 à 100 par machine) ; le déplacement se fait en arrière-plan, la table de routage est mise à jour **atomiquement**.

Routage

Comment un client trouve-t-il la bonne partition ? Un **routeur** (proxy), un client qui connaît la **table de routage** (récupérée d'un service de coordination), ou n'importe quel nœud qui redirige.

🧠 ***Astuce*** *: demande-toi toujours : quelle est ma requête la plus fréquente, et touche-t-elle une seule partition ? Une mauvaise clé transforme chaque requête en tempête sur tous les serveurs.*

Exemple pas à pas

Problème : mesurer la fraction de données déplacée en passant de 5 à 6 serveurs avec `hash mod N` et avec un anneau cohérent, l'effet des nœuds virtuels sur l'équilibre, un point chaud et son salage, et le nombre de partitions touchées par une requête sur une plage.

```
import bisect, hashlib, random
from collections import Counter

def h(valeur):
    return int(hashlib.md5(str(valeur).encode()).hexdigest()[:12], 16)

CLES = [f"inscription:{i}" for i in range(50000)]

# ---------------- 1. hash mod N : presque tout bouge ----------------
def mod_n(cle, n): return h(cle) % n
deplacees = sum(1 for c in CLES if mod_n(c, 5) != mod_n(c, 6)) / len(CLES)
print(f"hash mod N, de 5 à 6 serveurs : {deplacees:.1%} des clés déplacées (théori
   e : 5/6 = {5 / 6:.1%})")
assert abs(deplacees - 5 / 6) < 0.02
```

Exemple pas à pas (suite)

```
# ---------------- 2. anneau de hachage cohérent avec nœuds virtuels -------------
   ---
class Anneau:
    def __init__(self, serveurs, virtuels=200):
        self.virtuels, self.points = virtuels, []                # liste triée de
    (position, serveur)
        for s in serveurs: self.ajouter(s)
    def ajouter(self, serveur):
        for v in range(self.virtuels): bisect.insort(self.points, (h(f"{serveur}#{
   v}"), serveur))
    def retirer(self, serveur):
        self.points = [(p, s) for p, s in self.points if s != serveur]
    def trouver(self, cle):
        i = bisect.bisect(self.points, (h(cle), chr(0x10FFFF))) % len(self.points)
              # premier point dans le sens horaire
        return self.points[i][1]

anneau5 = Anneau([f"s{i}" for i in range(5)])
```

Exemple pas à pas (suite)

```
avant = {c: anneau5.trouver(c) for c in CLES}
anneau5.ajouter("s5")
apres = {c: anneau5.trouver(c) for c in CLES}
deplacees_c = sum(1 for c in CLES if avant[c] != apres[c]) / len(CLES)
print(f"anneau cohérent, de 5 à 6 serveurs : {deplacees_c:.1%} des clés déplacées
   (théorie : 1/6 = {1 / 6:.1%})")
assert abs(deplacees_c - 1 / 6) < 0.03
assert all(apres[c] == "s5" for c in CLES if avant[c] != apres[c])          # les
    clés déplacées vont TOUTES vers le nouveau serveur

# ---------------- 3. l'équilibre dépend des nœuds virtuels ----------------
def desequilibre(virtuels):
    a = Anneau([f"s{i}" for i in range(6)], virtuels)
    charge = Counter(a.trouver(c) for c in CLES)
    moyenne = len(CLES) / 6
    return max(charge.values()) / moyenne                                  # plus gros
    serveur / moyenne (1.0 = parfait)
print("nœuds virtuels | plus gros serveur / moyenne")
res = {}
```

Exemple pas à pas (suite)

```
for v in (1, 10, 100, 500):
    res[v] = desequilibre(v); print(f"      {v:4d}      |  {res[v]:.2f}")
assert res[1] > res[100] and res[500] < 1.15 and res[1] > 1.3

# ---------------- 4. un point chaud : le hachage équilibre les clés, pas le trafi
   c ----------------
rng = random.Random(1)
cours = [f"cours:{i}" for i in range(200)]
poids = [1 / (i + 1) ** 1.2 for i in range(200)]                          # quelques
   cours très populaires (loi de Zipf)
requetes = rng.choices(cours, weights=poids, k=100000)
a6 = Anneau([f"s{i}" for i in range(6)], 200)
charge = Counter(a6.trouver(c) for c in requetes)
print("trafic par serveur (sans salage) :", sorted(charge.values(), reverse=True))
ratio = max(charge.values()) / (len(requetes) / 6)
assert ratio > 1.5                                                          # un serv
   eur reçoit bien plus que sa part

def salage(cle, k=8):                                                       # on répa
```

Exemple pas à pas (suite)

```
   rtit une clé chaude sur k sous-clés
    return f"{cle}#{rng.randrange(k)}" if cle in {"cours:0", "cours:1", "cours:2"}
    else cle
charge_salee = Counter(a6.trouver(salage(c)) for c in requetes)
ratio_salee = max(charge_salee.values()) / (len(requetes) / 6)
print(f"plus gros serveur / moyenne : sans salage {ratio:.2f} → avec salage des 3
   clés les plus chaudes {ratio_salee:.2f}")
assert ratio_salee < ratio                                          # (le pri
   x : lire cette clé = interroger 8 sous-clés)

# ---------------- 5. requêtes sur une plage : hachage contre plages -------------
   ---
dates = [f"2025-{m:02d}-{j:02d}" for m in range(1, 13) for j in range(1, 29)]
       # clés ordonnées (336 jours)
frontieres = [f"2025-{m:02d}-01" for m in (3, 5, 7, 9, 11)]
      # 6 plages de 2 mois
def partition_plage(cle): return bisect.bisect(frontieres, cle)
requete = [d for d in dates if "2025-04-05" <= d <= "2025-04-25"]
     # « les inscriptions d'avril »
```

Exemple pas à pas (suite)

```
par_plage = {partition_plage(d) for d in requete}
par_hash = {int(h(d) % 6) for d in requete}
print(f"requête sur une plage de dates : partitions touchées → par plage {len(par_
   plage)}, par hachage {len(par_hash)}")
assert len(par_plage) == 1 and len(par_hash) == 6
# …mais les écritures du jour (clés monotones) vont toutes à la dernière plage
ecritures = Counter(partition_plage(d) for d in dates[-28:])
print("écritures du dernier mois par plage :", dict(ecritures), "← point chaud sur
    la dernière plage")
assert len(ecritures) == 1
```

Lecture :

1. `**hash mod N**` déplace ≈ 83 % des clés en passant de 5 à 6 serveurs. L'**anneau cohérent** n'en déplace que ≈ 17 % — et **toutes** vers le nouveau serveur (rien n'est brassé entre les anciens).
2. Avec **1 nœud virtuel** par serveur, le plus chargé a bien plus que sa part ; avec 100 à 500, l'équilibre devient bon.

Exemple pas à pas (suite)

3. Le **hachage équilibre les clés, pas le trafic** : sous une charge de type Zipf, un serveur reçoit 50 % de plus que la moyenne. **Saler** les clés chaudes répartit leurs requêtes, au prix d'une lecture éclatée sur k sous-clés.
4. **Plages contre hachage** : une requête sur « avril » touche 1 partition par plage, mais les 6 par hachage. En contrepartie, les clés monotones (dates) concentrent toutes les écritures récentes sur la dernière plage.

À toi, avec aide

Calcule par simulation la fraction de clés déplacées quand on **retire** un serveur d'un anneau de 8 serveurs (200 nœuds virtuels chacun) :

1. fraction déplacée (théorie : 1/8) ;
2. vers quels serveurs vont les clés déplacées ? Sont-elles réparties uniformément entre les 7 serveurs restants (avantage des nœuds virtuels) ou toutes vers un seul voisin ? Compare avec 1 seul nœud virtuel par serveur ;
3. quel est le plus gros reçu / la moyenne attendue ?

À toi, avec aide · réponse

Indices progressifs

Indice 1

Utilise la classe `Anneau` de l'exemple : `retirer`.

Indice 2

Compte les destinations des clés dont le serveur d'origine était le serveur retiré.

Indice 3

Avec 1 nœud virtuel, un seul voisin hérite de tout.

À toi, avec aide · réponse

Réponse

```
import bisect, hashlib
from collections import Counter

def h(v): return int(hashlib.md5(str(v).encode()).hexdigest()[:12], 16)
class Anneau:
    def __init__(self, serveurs, virtuels=200):
        self.virtuels, self.points = virtuels, []
        for s in serveurs:
            for v in range(virtuels): bisect.insort(self.points, (h(f"{s}#{v}"), s
   ))
    def retirer(self, s): self.points = [(p, x) for p, x in self.points if x != s]
    def trouver(self, c):
        i = bisect.bisect(self.points, (h(c), chr(0x10FFFF))) % len(self.points);
   return self.points[i][1]

CLES = [f"k{i}" for i in range(40000)]
for virtuels in (200, 1):
    a = Anneau([f"s{i}" for i in range(8)], virtuels)
    avant = {c: a.trouver(c) for c in CLES}
```

Réponse (suite)

```
    a.retirer("s3")
    apres = {c: a.trouver(c) for c in CLES}
    deplacees = [c for c in CLES if avant[c] != apres[c]]
    destinations = Counter(apres[c] for c in deplacees)
    print(f"{virtuels:3d} virtuels : {len(deplacees) / len(CLES):.1%} déplacées ;
destinations {sorted(destinations.values(), reverse=True)}")
    if virtuels == 200:
        assert abs(len(deplacees) / len(CLES) - 1 / 8) < 0.03 and len(destinations
) == 7 and max(destinations.values()) < 0.3 * len(deplacees)
    else:
        assert len(destinations) <= 2                  # un seul voisin hérite q
uasiment de tout
```

Avec 200 nœuds virtuels, la charge du serveur retiré se répartit sur les **7** autres (le plus gros reçoit moins de 30 % du déplacement). Avec 1 seul nœud virtuel, la part du serveur retiré est très inégale (ici minuscule : le tirage est aléatoire) et **un seul voisin** reçoit tout : il risque de tomber à son tour (panne en cascade). C'est la raison de l'usage des nœuds virtuels.

À toi, en solo

Partitionnement par plages avec scission automatique. Écris `TablePlages` :

1. une liste de plages triées, chacune avec une capacité maximale (par exemple 1 000 enregistrements) ;
2. `ecrire(cle, valeur)` route vers la bonne plage ; si une plage dépasse la capacité, elle est **scindée** en deux au milieu (médiane des clés) ;
3. `lire_plage(a, b)` renvoie les valeurs et le nombre de plages touchées ;
4. teste avec des clés **aléatoires** (équilibre naturel) puis avec des clés **monotones** (horodatages croissants) : compte les écritures reçues par chaque plage ; montre que les clés monotones concentrent le trafic sur la dernière plage (point chaud) ;
5. corrige en préfixant la clé par un **compartiment de hachage** (`h(cle) mod 8` + clé) : mesure la répartition des écritures et le nombre de plages touchées par une lecture de plage (compromis).

À toi, en solo · réponse

Pistes de correction

```
import bisect, hashlib, random
from collections import Counter

def h(v): return int(hashlib.md5(str(v).encode()).hexdigest()[:8], 16)

class TablePlages:
    def __init__(self, capacite=500):
        self.capacite = capacite
        self.bornes = []                                    # bornes inférieure
   s des plages 1..n (la plage 0 démarre à -∞)
        self.plages = [{}]                                  # chaque plage : di
   ct cle -> valeur
        self.ecritures = Counter()                          # écritures reçues
   par numéro de plage (au moment de l'écriture)
    def _indice(self, cle): return bisect.bisect(self.bornes, cle)
    def ecrire(self, cle, valeur):
        i = self._indice(cle); self.plages[i][cle] = valeur; self.ecritures[i] +=
   1
        if len(self.plages[i]) > self.capacite: self._scinder(i)
```

Pistes de correction (suite)

```
    def _scinder(self, i):
        cles = sorted(self.plages[i]); milieu = cles[len(cles) // 2]
        gauche = {k: v for k, v in self.plages[i].items() if k < milieu}
        droite = {k: v for k, v in self.plages[i].items() if k >= milieu}
        self.plages[i:i + 1] = [gauche, droite]; self.bornes.insert(i, milieu)
        # recalcul des compteurs : on re-indexe simplement
        nouveau = Counter()
        for j, c in self.ecritures.items(): nouveau[j if j < i else j + 1] += c
        nouveau[i] += 0; self.ecritures = nouveau
    def lire_plage(self, a, b):
        debut, fin = self._indice(a), self._indice(b)
        valeurs = [v for j in range(debut, fin + 1) for k, v in sorted(self.plages
   [j].items()) if a <= k <= b]
        return valeurs, fin - debut + 1

def repartition(cles):
    t, vers_derniere = TablePlages(500), 0
    for n, c in enumerate(cles):
        if n >= len(cles) // 2 and t._indice(c) == len(t.plages) - 1: vers_dernier
```

Pistes de correction (suite)

```
    e += 1      # (régime établi) cette écriture vise-t-elle la DERNIÈRE plage ?
         t.ecrire(c, n)
     return t, vers_derniere / (len(cles) - len(cles) // 2)

rng = random.Random(1)
aleatoires = [f"{rng.randrange(10**9):09d}" for _ in range(10000)]
monotones = [f"2025-{1 + n // 30000:02d}-{1 + (n // 1000) % 28:02d}T{n % 1000:04d}
    " for n in range(10000)]
t_a, part_a = repartition(aleatoires); t_m, part_m = repartition(monotones)
print(f"clés aléatoires : {len(t_a.plages):3d} plages ; {part_a:6.1%} des écriture
    s du régime établi visent la dernière plage")
print(f"clés monotones  : {len(t_m.plages):3d} plages ; {part_m:6.1%} des écriture
    s visent la dernière plage  ← point chaud")
assert part_m > 0.95 and part_a < 0.10

# correction : compartiment de hachage en préfixe
compartimentees = [f"{h(c) % 8}|{c}" for c in monotones]
t_c, part_c = repartition(compartimentees)
print(f"avec préfixe de hachage : {len(t_c.plages)} plages ; {part_c:.1%} des écri
```

À toi, en solo · réponse

Pistes de correction (suite)

```
   tures visent la dernière plage")
assert part_c < 0.25
# …mais une lecture d'un intervalle de dates doit maintenant interroger les 8 comp
   artiments
_, t_touchees = t_c.lire_plage("0|2025-01-01", "0|2025-01-28T9999")
print("plages touchées pour lire un seul compartiment :", t_touchees, "(il en faut
    8 fois plus pour couvrir l'intervalle de dates complet)")
```

Le **préfixe de hachage** répartit les écritures sur 8 groupes (le point chaud disparaît), mais une lecture « par date » doit désormais interroger les **8 compartiments** puis fusionner. C'est le compromis fondamental : l'équilibre des écritures contre la simplicité des lectures par plage.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
`hash(clé) mod N` avec N variable	83 % des données déplacées à chaque agrandissement	Hachage cohérent, ou beaucoup de partitions fixes
Clé monotone (horodatage) en partitionnement par plage	Toutes les écritures sur la dernière partition	Préfixe de hachage ou clé composite
Supposer qu'un hachage uniforme équilibre le trafic	Une clé populaire surcharge une partition	Cache, réplicas de lecture, salage de la clé chaude
Choisir la clé sans connaître les requêtes	Chaque requête interroge toutes les partitions	Regrouper ce qui est accédé ensemble

Mini-boss

Mini-boss

Magasin partitionné de HashBox. Construis un magasin clé-valeur partitionné avec :

1. un anneau cohérent à nœuds virtuels, un routeur avec **table de routage versionnée** ;
2. ajout et retrait de nœuds avec **migration** des clés concernées, sans perte pendant la migration (les lectures consultent l'ancienne et la nouvelle position tant que la migration n'est pas terminée) ;
3. détection automatique d'un **point chaud** (une partition reçoit plus de 2× la moyenne) et salage automatique de la clé responsable ;
4. requête de type **scatter-gather** (comptage global) avec délai d'attente par partition et résultat partiel signalé ;
5. simulation de 10 000 clés, charge Zipf, ajout de 2 nœuds puis panne d'un nœud ;
6. tableau : fraction déplacée à chaque événement, équilibre avant/après salage, latence du scatter-gather (la plus lente des partitions) ;

Mini-boss

Mini-boss (suite)

7. 10 tests (aucune clé perdue, déplacement ≤ 1/N + marge, équilibre).

Mini-boss · réponse

Pistes de correction

Un bon travail : classe `Anneau` + classe `Magasin` avec `table_de_routage[version]`, migration en deux temps (copie puis bascule), lecture « double » pendant la migration; détecteur de point chaud par comptage glissant; salage automatique avec registre des clés salées (lectures éclatées); scatter-gather avec délais et résultat partiel (`{"resultat": n, "partitions_manquantes": [...]}`); tests des invariants (aucune clé perdue, déplacement proche de 1/N, ratio d'équilibre < 1,2 avec 200 nœuds virtuels).

Quiz éclair

Quiz éclair

1. Avec `hash(clé) mod N`, ajouter un serveur :

A. ne déplace aucune clé
B. déplace la majorité des clés
C. déplace exactement 1/N des clés

2. Le hachage cohérent avec nœuds virtuels :

A. supprime le besoin de répliquer
B. empêche toute panne
C. limite les déplacements à environ 1/(N+1) des clés et équilibre la charge

3. Une requête sur une plage de clés est efficace avec :

A. un partitionnement par plage, qui touche peu de partitions

Quiz éclair

Quiz éclair (suite)

B. un partitionnement par hachage, qui en touche beaucoup
C. aucun partitionnement possible

4. 🔁 *Révision mission 05* : En réplication asynchrone, quand le leader tombe :

A. aucune écriture n'est jamais perdue
B. des écritures déjà confirmées au client peuvent être perdues
C. les suiveurs sont toujours à jour

5. 🔁 *Révision mission 05* : La règle de quorum qui garantit de lire au moins un réplica à jour est :
A. $R + W < N$
B. $R = W = 1$
C. $R + W > N$

Quiz éclair · réponse

Réponses

1. **B**.
2. **C**.
3. **A**.
4. **B**.
5. **C**.

Je réfléchis

1. Quelle est ta requête la plus fréquente, et sur combien de partitions s'exécuterait-elle ?
2. Quelle clé de ton système est trop populaire pour rester sur une seule machine ?

À retenir

À retenir

1. Partitionnement = diviser ; réplication = copier ; on combine.
2. Hachage (équilibre) contre plage (requêtes d'intervalles) ; clés monotones = point chaud.
3. `hash mod N` déplace presque tout ; l'anneau cohérent déplace ≈ 1/(N+1).
4. Nœuds virtuels : équilibre et répartition de la charge d'un nœud perdu.
5. Point chaud : cache, réplicas, salage ; scatter-gather coûteux.

MISSION ACCOMPLIE

+100 XP

Badge débloqué : Découpeur de données

Prochaine mission : Cohérence et consensus : panorama

