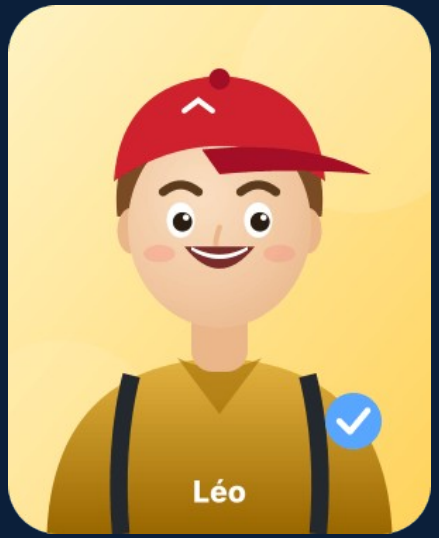
HEF-130 · MISSION 08 / 10

Cache et invalidation



Un cache est le moyen le plus puissant d'accélérer un système — et la source la plus fréquente de bugs étranges. « Il n'y a que deux choses difficiles en informatique : l'invalidation de cache et nommer les choses. »

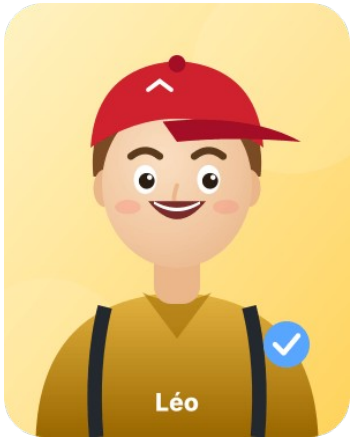


Badge : Gardien de la mémoire rapide

105 min · avancé

L'histoire

HashBox met en cache la liste des cours pour soulager la base. Les temps de réponse passent de 120 ms à 4 ms. Mais le lendemain, une formatrice modifie un cours, et ses étudiants continuent de voir l'ancienne version pendant une heure. Une autre fois, à midi, le cache expire en même temps pour tout le monde : 2 000 requêtes identiques se ruent sur la base, qui s'effondre.



> *« Un cache, c'est une copie, dit Sam. Une copie peut devenir fausse. Il y a donc trois questions à se poser : que garde-t-on, jusqu'à quand, et comment sait-on que c'est périmé ? Et une quatrième : que se passe-t-il quand le cache est vide et que tout le monde arrive en même temps ? »*

Tu apprends à implémenter des politiques d'éviction (LRU, LFU), à mesurer un taux de succès, à choisir une durée de vie et une stratégie d'invalidation, et à te protéger de la ruée (*cache stampede*).

Ta mission

À la fin, je serai capable de…

1. calculer le temps d'accès moyen à partir du taux de succès (hit ratio)
2. implémenter et comparer des politiques d'éviction (FIFO, LRU, LFU) sur une charge réaliste
3. choisir entre durée de vie, invalidation explicite et clés versionnées
4. empêcher la ruée sur la source avec un verrou par clé (*single-flight*)
5. distinguer cache-aside, écriture immédiate (write-through) et différée (write-back)

Défi d'échauffement

Défi d'échauffement

Tu gardes dans ta poche les numéros de téléphone que tu composes souvent, plutôt que de chercher dans l'annuaire. Que se passe-t-il quand un ami change de numéro ? Que fais-tu quand ta poche est pleine et que tu veux ajouter un numéro ? Comment sais-tu quels numéros garder ?

Le concept

Pourquoi un cache

Un **cache** garde une **copie** de données coûteuses à obtenir, dans un endroit plus rapide. Il est partout : registres et caches du processeur, cache de pages du système, navigateur, CDN, cache applicatif (Redis, Memcached), cache de requêtes de la base.

```
temps d'accès moyen = h × t_cache + (1 − h) × t_source        (h = taux de succès)
```

Avec `t_cache = 1 ms`, `t_source = 100 ms` : à h = 50 %, 50,5 ms ; à 90 %, **10,9 ms** ; à 99 %, **2 ms**. Chaque point de taux de succès compte énormément quand la source est très lente.

Patrons d'utilisation

Le concept

Patrons d'utilisation (suite)

Patron	Lecture	Écriture
Cache-aside (le plus courant)	l'application regarde le cache ; sinon va à la source et remplit le cache	l'application écrit la source, puis **invalide** (ou met à jour) le cache
Read-through	le cache lui-même charge depuis la source	idem
Write-through	—	écriture **dans le cache et la source** ensemble
Write-back	—	écriture dans le cache, source mise à jour **plus tard** (rapide, risque de perte)

Éviction : que retirer quand c'est plein ?

Politique	Retire	Bon pour
FIFO	le plus ancien arrivé	simple, souvent médiocre

Le concept

Eviction : que retirer quand c'est plein ? (suite)

Politique	Retire	Bon pour
LRU (*Least Recently Used*)	le moins récemment utilisé	localité temporelle : usage général
LFU (*Least Frequently Used*)	le moins fréquemment utilisé	popularité stable
TTL	ce qui a expiré	données qui vieillissent
Aléatoire	un au hasard	surprenamment correct, peu de coût

Les accès réels suivent souvent une loi de type **Zipf** : quelques éléments très populaires, une longue traîne rare. Un petit cache (5 % des données) peut alors couvrir 70 à 90 % des accès.

Invalider : le vrai problème

Stratégie	Principe	Risque
Durée de vie (TTL)	l'entrée expire après N secondes	données périmées jusqu'à N s

Le concept

Invalider : le vrai problème (suite)

Stratégie	Principe	Risque
Invalidation explicite	on supprime l'entrée à chaque écriture	oublier un chemin d'écriture
Clés versionnées	la clé contient un numéro de version : `cours:42:v7` ; changer la version rend l'ancienne entrée inatteignable	accumulation d'anciennes entrées (éviction)
Invalidation par étiquettes	à chaque entrée, des étiquettes (`cours:42`, `categorie:algo`) ; on invalide par étiquette	complexité
Événements	la source publie « changé » et les caches se mettent à jour	délai, perte de messages

Combiner **TTL + invalidation** : le TTL est le filet de sécurité qui borne l'erreur si une invalidation est manquée.

Le concept

Clés de cache

La clé doit contenir **tout ce qui change le résultat** : identifiant, langue, droits de l'utilisateur, version de l'API. Une clé trop petite mélange des résultats (fuite entre utilisateurs !) ; trop grande, elle ne réutilise rien.

La ruée (*cache stampede*)

Quand une entrée populaire expire, **toutes** les requêtes concurrentes la trouvent absente et vont toutes à la source. Remèdes :

- **Un seul chargeur par clé** (*single-flight*, verrou) : les autres attendent son résultat ;
- **Expiration avec gigue** : TTL = 300 s ± 10 % pour éviter l'expiration simultanée ;
- **Rafraîchissement anticipé** ou **servir l'ancien pendant le rechargement** (*stale-while-revalidate*) ;

Le concept

La ruée (*cache stampede*) (suite)

- **Mise en cache des absences** (cache négatif) pour éviter de marteler la source avec des clés inexistantes.

Cohérence

Un cache fait renoncer à la cohérence **parfaite** : il y a une fenêtre où une lecture peut voir l'ancienne valeur. À décider consciemment : pour un solde bancaire, non ; pour la liste des cours, quelques secondes d'ancienneté sont acceptables.

Mesurer

Taux de succès, taux d'éviction, taille, latence avec et sans cache, **âge** des données servies. Un cache dont le taux de succès est de 5 % ne sert à rien et ajoute de la complexité.

Le concept

Mesurer (suite)

🧠 ***Astuce*** *: demande-toi, avant d'ajouter un cache : quelle est l'erreur acceptable (ancienneté) ? qui invalide ? que se passe-t-il à froid ? Si tu ne peux pas répondre, tu n'es pas prêt à le mettre en production.*

Exemple pas à pas

Problème : implémenter FIFO, LRU et LFU, les comparer sur une charge Zipf, voir l'effet de la taille sur le taux de succès, la durée de vie (TTL) avec horloge simulée, et empêcher la ruée avec un verrou par clé.

```
import collections, random, threading, time

# ---------------- politiques d'éviction ----------------
class Cache:
    def __init__(self, taille): self.taille, self.succes, self.echecs = taille, 0,
    0
    def acces(self, cle):
        if self._present(cle):
            self.succes += 1; self._toucher(cle); return True
        self.echecs += 1; self._inserer(cle); return False
    @property
    def taux(self): return self.succes / (self.succes + self.echecs)

class FIFO(Cache):
```

Exemple pas à pas

Exemple pas à pas (suite)

```
    def __init__(self, t): super().__init__(t); self.d = collections.OrderedDict()
    def _present(self, c): return c in self.d
    def _toucher(self, c): pass
    def _inserer(self, c):
        self.d[c] = True
        if len(self.d) > self.taille: self.d.popitem(last=False)

class LRU(Cache):
    def __init__(self, t): super().__init__(t); self.d = collections.OrderedDict()
    def _present(self, c): return c in self.d
    def _toucher(self, c): self.d.move_to_end(c)
    def _inserer(self, c):
        self.d[c] = True
        if len(self.d) > self.taille: self.d.popitem(last=False)

class LFU(Cache):
    def __init__(self, t): super().__init__(t); self.d, self.freq = {}, collection
   s.Counter()
    def _present(self, c): return c in self.d
```

Exemple pas à pas (suite)

```
    def _toucher(self, c): self.freq[c] += 1
    def _inserer(self, c):
        if len(self.d) >= self.taille:
            moins = min(self.d, key=lambda k: self.freq[k]); del self.d[moins]; de
   l self.freq[moins]
        self.d[c] = True; self.freq[c] = 1

def charge_zipf(n_cles=10000, n_acces=60000, s=1.1, graine=1):
    rng = random.Random(graine)
    poids = [1 / (i + 1) ** s for i in range(n_cles)]
    return rng.choices(range(n_cles), weights=poids, k=n_acces)

ACCES = charge_zipf()
print("taille du cache | FIFO     LRU      LFU")
resultats = {}
for taille in (50, 200, 1000, 3000):
    ligne = {}
    for nom, classe in (("FIFO", FIFO), ("LRU", LRU), ("LFU", LFU)):
        c = classe(taille)
```

Exemple pas à pas (suite)

```
        for k in ACCES: c.acces(k)
        ligne[nom] = c.taux
    resultats[taille] = ligne
    print(f"  {taille:5d} ({taille / 10000:5.1%})  | {ligne['FIFO']:6.1%}  {ligne[
   'LRU']:6.1%}  {ligne['LFU']:6.1%}")
assert resultats[200]["LRU"] > resultats[200]["FIFO"] and resultats[200]["LFU"] >=
    resultats[200]["LRU"] - 0.02
assert resultats[200]["LRU"] > 0.5                                  # un cache de
   2 % des données sert plus de la moitié des accès

def temps_moyen(h, t_cache=1.0, t_source=100.0): return h * t_cache + (1 - h) * t_
   source
print("temps d'accès moyen à h = 50 %, 90 %, 99 % :", [round(temps_moyen(h), 1) fo
   r h in (0.5, 0.9, 0.99)])
assert [round(temps_moyen(h), 1) for h in (0.5, 0.9, 0.99)] == [50.5, 10.9, 2.0]

# ---------------- TTL avec horloge simulée ----------------
class CacheTTL:
    def __init__(self, ttl, horloge): self.ttl, self.horloge, self.d = ttl, horlog
```

Exemple pas à pas (suite)

```
   e, {}
    def lire(self, cle, charger):
        entree = self.d.get(cle)
        if entree and self.horloge() < entree[1]:
            return entree[0], "cache"
        valeur = charger(cle); self.d[cle] = (valeur, self.horloge() + self.ttl)
        return valeur, "source"

t = {"now": 0.0}; source = {"cours": "v1"}
cache = CacheTTL(60, lambda: t["now"])
assert cache.lire("cours", source.get) == ("v1", "source")
source["cours"] = "v2"                                          # la source c
   hange…
t["now"] = 30; assert cache.lire("cours", source.get) == ("v1", "cache")        #
   …mais le cache sert l'ancienne valeur : erreur bornée par le TTL
t["now"] = 61; assert cache.lire("cours", source.get) == ("v2", "source")

# ---------------- la ruée, avec et sans verrou par clé ----------------
def ruee(avec_verrou, threads=20):
```

Exemple pas à pas

Exemple pas à pas (suite)

```
    appels, cache, verrous, g = [0], {}, collections.defaultdict(threading.Lock),
threading.Lock()
    barriere = threading.Barrier(threads)
    def charger():
        appels[0] += 1; time.sleep(0.05); return "valeur"            # la source
est lente
    def lire(cle):
        if cle in cache: return cache[cle]
        if not avec_verrou:
            cache[cle] = charger(); return cache[cle]
        with g: verrou = verrous[cle]
        with verrou:                                                  # un seul c
hargeur par clé ; les autres attendent
            if cle not in cache: cache[cle] = charger()
            return cache[cle]
    def tache(): barriere.wait(); lire("cours:42")
    ts = [threading.Thread(target=tache) for _ in range(threads)]
    for x in ts: x.start()
    for x in ts: x.join()
```

Exemple pas à pas (suite)

```
    return appels[0]

sans, avec = ruee(False), ruee(True)
print(f"20 requêtes simultanées sur une clé absente : appels à la source sans prot
   ection = {sans}, avec verrou par clé = {avec}")
assert avec == 1 and sans >= 10
```

Lecture :

1. Sur une charge **Zipf**, un cache de seulement **2 %** des données (200 clés sur 10 000) sert **plus de la moitié** des accès avec LRU ; LRU et LFU battent nettement FIFO. Plus on agrandit le cache, plus le gain diminue (rendements décroissants).
2. Le **temps moyen** montre pourquoi chaque point de taux de succès compte : 50 ms à 50 %, 11 ms à 90 %, 2 ms à 99 %.
3. Le **TTL** borne l'erreur : après un changement à la source, le cache peut servir l'ancienne valeur pendant au plus 60 secondes.

Exemple pas à pas (suite)

4. Sans protection, la ruée envoie **plusieurs appels simultanés** à la source ; avec un verrou par clé, **un seul** chargeur travaille et les autres attendent son résultat.

À toi, avec aide

Un service a un temps de lecture à la source de **80 ms** et un cache à **2 ms**. Le trafic est de **500 requêtes/s**.

1. Calcule le temps moyen et la charge sur la source pour un taux de succès de 70 %, 90 % et 98 %.
2. La source supporte au maximum **100 requêtes/s**. Quel taux de succès minimal faut-il pour que la source tienne ?
3. Après un redéploiement, le cache est **vide** et le taux de succès tombe à 0 % pendant quelques minutes. Que se passe-t-il, et quelle parade proposes-tu (au moins deux) ?

À toi, avec aide · réponse

Indices progressifs

Indice 1

Charge sur la source = trafic × (1 − h).

Indice 2

Pour que la source tienne : 500 × (1 − h) ≤ 100.

Indice 3

À froid, la source reçoit 500 requêtes/s alors qu'elle n'en supporte que 100.

Réponse

```
for h in (0.70, 0.90, 0.98):
    print(f"h = {h:.0%} : temps moyen {h * 2 + (1 - h) * 80:5.1f} ms | charge sur
   la source {500 * (1 - h):5.0f} req/s")
h_min = 1 - 100 / 500
print(f"taux de succès minimal : {h_min:.0%}")
assert round(h_min, 2) == 0.80
```

1. 70 % : 25,4 ms, **150 req/s** (la source est déjà dépassée) ; 90 % : 9,8 ms, **50 req/s** ; 98 % : 3,6 ms, **10 req/s**.
2. Il faut au moins **80 %** de succès pour que la source tienne ses 100 req/s.
3. Au démarrage à froid, la source reçoit **500 req/s pour 100 supportables** : files d'attente, timeouts, effondrement. Parades : **préchauffer** le cache avant de basculer le trafic ; **monter en charge progressivement** (canary, HEF-120) ; **verrou par clé** (single-flight) pour réduire les doublons ; **limiter** le débit vers la source avec dégradation (réponses de repli) ; **cache persistant ou partagé** (Redis) qui survit au redéploiement.

À toi, en solo

Cache à invalidation par étiquettes, avec lecture cohérente. Implémente `CacheEtiquetes` :

1. `lire(cle, chargeur, etiquettes, ttl)` : cache-aside ; chaque entrée porte des étiquettes (`cours:42`, `formateur:7`) ;
2. `invalider(etiquette)` supprime toutes les entrées associées ;
3. TTL avec **gigue** (±10 %) calculée par un générateur à graine fixée ;
4. verrou par clé contre la ruée ;
5. **cache négatif** : une clé absente de la source est mémorisée brièvement ;
6. métriques : succès, échecs, invalidations, âge moyen des données servies ;
7. test de **cohérence** : une écriture suivie d'une invalidation ne permet plus de lire l'ancienne valeur (sur un scénario de 1000 opérations aléatoires avec graine fixée, comparer avec un modèle sans cache) — il ne doit y avoir aucune lecture périmée, si les écritures invalident toujours.

À toi, en solo · réponse

Pistes de correction

```
import collections, random, threading

class CacheEtiquettes:
    def __init__(self, horloge, graine=1):
        self.h, self.rng = horloge, random.Random(graine)
        self.d, self.index, self.verrou = {}, collections.defaultdict(set), thread
   ing.RLock()
        self.succes = self.echecs = self.invalidations = 0
        self.ages = []
    def lire(self, cle, chargeur, etiquettes=(), ttl=60.0, ttl_absent=5.0):
        with self.verrou:                                                  # verro
   u global simple : un seul chargeur à la fois
            e = self.d.get(cle)
            if e and self.h() < e["expire"]:
                self.succes += 1; self.ages.append(self.h() - e["cree"]); return e
   ["valeur"]
            self.echecs += 1
            valeur = chargeur(cle)
            duree = (ttl if valeur is not None else ttl_absent) * (1 + self.rng.un
```

À toi, en solo · réponse

Pistes de correction (suite)

```
   iform(-0.1, 0.1))     # gigue
            self.d[cle] = {"valeur": valeur, "cree": self.h(), "expire": self.h()
   + duree, "etiquettes": set(etiquettes)}
            for t in etiquettes: self.index[t].add(cle)
            return valeur
    def invalider(self, etiquette):
        with self.verrou:
            for cle in self.index.pop(etiquette, set()):
                if self.d.pop(cle, None) is not None: self.invalidations += 1

t = {"now": 0.0}
source = {f"cours:{i}": f"v0-{i}" for i in range(20)}
cache = CacheEtiquettes(lambda: t["now"])
chargements = []
def charger(cle): chargements.append(cle); return source.get(cle)

rng = random.Random(7); lectures_perimees = 0
for op in range(1000):
    t["now"] += rng.uniform(0, 2)
```

À toi, en solo · réponse

Pistes de correction (suite)

```
    i = rng.randrange(25)                                                  # 5 clé
   s n'existent pas (cache négatif)
    cle = f"cours:{i}"
    if rng.random() < 0.15 and cle in source:                              # écrit
   ure : on modifie la source PUIS on invalide
        source[cle] = f"v{op}-{i}"
        cache.invalider(cle)
    else:
        lu = cache.lire(cle, charger, etiquettes=(cle, "catalogue"), ttl=60)
        if lu != source.get(cle):
            lectures_perimees += 1
print(f"succès {cache.succes}, échecs {cache.echecs}, invalidations {cache.invalid
   ations}, âge moyen servi {sum(cache.ages) / len(cache.ages):.1f} s, chargements
    {len(chargements)}")
assert lectures_perimees == 0                                              # cohér
   ence : jamais de valeur périmée si chaque écriture invalide
assert cache.succes > cache.echecs and len(chargements) == cache.echecs
cache.lire("cours:1", charger, ("cours:1", "catalogue")); cache.invalider("catalog
   ue")
```

Pistes de correction (suite)

```
n_avant = len(chargements); cache.lire("cours:1", charger, ("cours:1", "catalogue"
   ))
assert len(chargements) == n_avant + 1                                  # l'inv
   alidation par étiquette a bien vidé l'entrée
```

Le **TTL** n'est ici qu'un filet de sécurité : la cohérence vient de l'**invalidation à chaque écriture**. Le test aléatoire avec modèle de référence est la bonne façon de valider un cache : on compare ce qu'il renvoie à ce que renverrait la source.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Aucune stratégie d'invalidation	Données périmées sans limite	TTL + invalidation explicite
Clé de cache incomplète	Mélange de résultats entre utilisateurs ou langues	Inclure tout ce qui change le résultat
Expiration simultanée	Ruée vers la source	Gigue, single-flight, rafraîchissement anticipé
Cache dont le taux de succès est faible	Complexité sans gain	Mesurer ; supprimer s'il ne sert pas

Mini-boss

Couche de cache pour le catalogue de HashBox. Conçois et teste une couche de cache pour le catalogue (cours, formateurs, prix) avec : 200 000 lectures/jour pour 5 000 cours (loi de Zipf), 50 modifications/jour, une source à 90 ms.

1. choisis la politique d'éviction et la taille par simulation (taux de succès contre taille) ;
2. conçois les clés (langue, devise, droits) et les étiquettes d'invalidation ;
3. implémente TTL avec gigue, invalidation par étiquettes, single-flight, cache négatif ;
4. simule une journée avec horloge virtuelle : taux de succès, charge sur la source, âge maximal servi ;
5. simule un redéploiement à froid : mesure le pic sur la source avec et sans préchauffage ;
6. écris les tests de cohérence contre un modèle sans cache ;
7. définis les alertes (taux de succès, âge, taux d'éviction) et le budget d'ancienneté accepté.

Mini-boss · réponse

Pistes de correction

Un bon travail : simulation Zipf pour choisir LRU et une taille couvrant ~90 % des accès avec ~10 % des cours; invalidation par étiquettes à chaque écriture (`cours:ID`, `formateur:ID`, `catalogue`), TTL de 10 minutes ± 10 %, verrou par clé; tableau de la charge source avec et sans préchauffage (le préchauffage supprime le pic de démarrage); test aléatoire contre un modèle de référence prouvant l'absence de lecture périmée; alertes sur taux de succès < 80 %, âge servi > TTL, évictions anormales.

Quiz éclair

1. Le temps d'accès moyen d'un cache dépend de :

A. uniquement de sa taille
B. de la couleur des données
C. son taux de succès et des temps du cache et de la source

2. Une politique LRU retire :

A. l'entrée la moins récemment utilisée
B. l'entrée la plus grande
C. une entrée au hasard uniquement

3. La ruée sur le cache (cache stampede) se prévient par :

A. l'augmentation du nombre de requêtes

Quiz éclair

Quiz éclair (suite)

B. un verrou par clé (un seul chargeur) et une expiration avec gigue
C. la suppression du cache

4. 🔁 *Révision mission 07* : Un `SCAN` dans un plan d'exécution signifie :

A. une recherche par index
B. une erreur de syntaxe
C. un parcours complet de la table

5. 🔁 *Révision mission 07* : Le problème N+1 correspond à :

A. une requête pour la liste puis une requête par élément
B. une table avec N+1 colonnes
C. un index en trop

Quiz éclair · réponse

Réponses

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

Je réfléchis

1. Quelle donnée mise en cache dans ton système pourrait être périmée, et quelle ancienneté serait acceptable ?
2. Que se passerait-il demain si ton cache était vidé d'un coup pendant l'heure de pointe ?

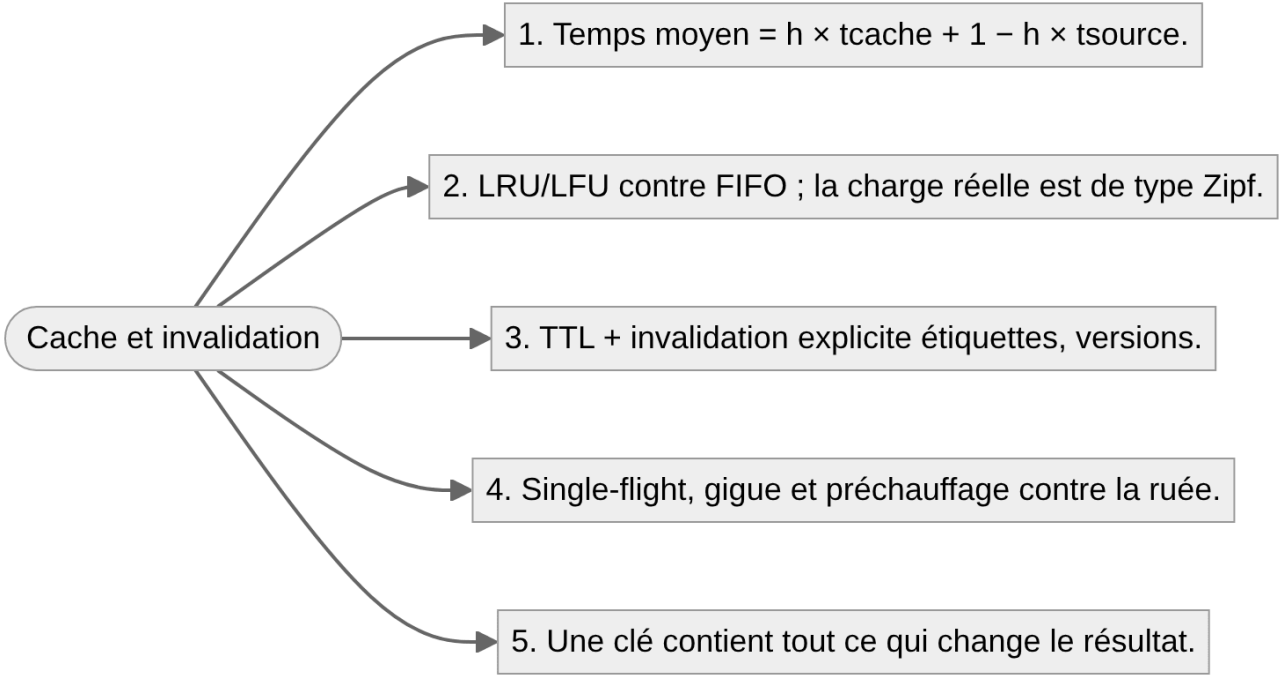
À retenir

À retenir

1. Temps moyen = h × t_cache + (1 − h) × t_source.
2. LRU/LFU contre FIFO ; la charge réelle est de type Zipf.
3. TTL + invalidation explicite (étiquettes, versions).
4. Single-flight, gigue et préchauffage contre la ruée.
5. Une clé contient tout ce qui change le résultat.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Gardien de la mémoire rapide

Prochaine mission : Concurrence et charge

