

Technologies de cache

Web / applicatif

Emmanuel Coquery

Caches

- Objectif: améliorer les performances
- Principe: sauvegarder le résultat d'un traitement de façon à ne pas refaire ce traitement lors de futures accès
- Utilisables à de nombreux niveaux: du cache CPU L1 au CDNs

Concepts généraux

- Granularité:
 - Unité d'information transmise entre le client et le cache (e.g. CPU-L1: mot)
 - Unité d'information transmise entre le cache et la mémoire de niveau supérieur (e.g. L1-L2: ligne)
- Localité:
 - Spatiale: données « proches » accédées ensemble
 - Temporelle: traitements sur une même donnée proches dans le temps
- Clé identifiant les données (e.g. adresse mémoire pour un cache physique, numéro de compte client pour un cache applicatif)

Les caches sont partout

- Caches matériels: CPU, disques durs
- Caches système: fichiers
- Caches internes aux applications:
 - caches de tables/requêtes dans les SGBD
 - caches d'objets dans les ORMs
(e.g. EntityManager en Java)
- Caches Web: (reverse)-proxys, cache navigateur, CDNs

Caches « applicatifs »

- Données résultant d'une combinaison d'accès aux données et de calculs
 - Résultats de rendu de page
 - architectures type CGI ou PHP
 - pages lourdes (gros volume, génération de HTML complexe)
 - Extraction de données complexes (e.g. profil utilisateur)
 - Requêtes multiples
 - Plusieurs sources de données à interroger

Pertinence des données en cache

- Surcoûts potentiels liés au cache
 - Temps de calcul en cas défaut de cache
 - Normalement peu élevé
- Occupation mémoire/disque
 - En général pas toutes les données en cache
 - 80/20:
 - 80% des demandes correspondent à
 - 20% des données

Cohérences des données en cache

- Modification des données de niveau supérieur
 - Invisible depuis le cache (cache statique)
 - Invalide le cache
 - e.g. supprime la donnée modifiée du cache
- Met à jour le cache (cache dynamique)
 - write-through: en simultané avec les données de niveau supérieur
 - write-back: lors de la suppression de la donnée depuis le cache

Gestion des modifications invisibles depuis le cache

- Durée de vie associée à la donnée
 - dépend du métier
 - globale vs par type d'objet
- TTL (Time To Live) vs TTI (Time To Idle)

Invalidation des données en cache

- Simple dans le cas de données atomiques / identifiées par la clé
- Complexe en cas de modification de données indirectement référencées
 - e.g. cache d'une page listant les produits correspondant à un mot clé (clé = mot) à invalider en cas de modification d'un mot clé
 - surcoût potentiel de recherche des conséquences de la modification

Mise à jour du cache en write-through

- Mêmes problèmes que pour l'invalidation de cache
- Rechargement des données
 - Les données seront-elles utilisées dans un temps court (avant d'être supprimées) ?
 - Peut être moins coûteux qu'une suppression puis une réinsertion à la demande
 - si valeur modifiée disponible
 - ou si on veut éviter une latence à la prochaine lecture

Politique d'éviction

- Si pas assez de place pour ajouter une donnée dans le cache, quelle donnée supprimer ?
- Mesures classiques:
 - Most/Least Recently Used (MRU)
 - Most/Least Frequently Used (MFU)
 - Liées au TTL / TTI

Initialisation

- Augmentation du temps de latence si une donnée n'est pas en cache
- Sollicitation élevée de la mémoire de niveau supérieur (e.g. SGBD/serveur Web)
- Potentiellement problématique:
 - Lors de la mise en place de l'application
 - Lors d'un redémarrage
- Pré-remplissage du cache
 - En particulier en cas de redémarrage
 - Nécessite une forme de persistance

Cache et HTTP

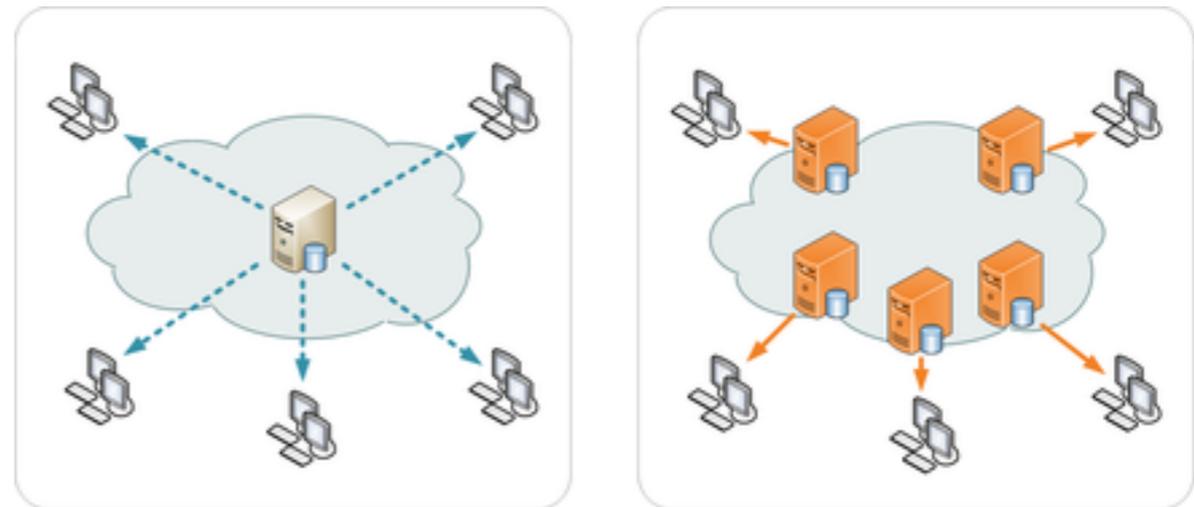
- Directives de gestion du cache
 - Fraîcheur des données:
 - Cache-Control: max-age
 - Expires
 - (In)validation:
 - Last-Modified
 - ETag (tag de version)
- Pas forcément utile si le cache est dynamique (rmq: le cache client est en général statique)

Cache par fragment

- Cache de parties de page Web
- Utile lors de la réutilisation de nombreux éléments aux sein d'une grande variété de pages
 - Personnalisations utilisateur: mélanges parties communes et parties personnalisées au sein d'une même page
 - Portails Web
- Assemblage possible côté serveur ou côté client

CDN: Content Delivery Networks

- Réseau de cache distribués géographiquement
- Réduction du temps de latence et augmentation du débit
- Plutôt du contenu statique (images, scripts js, fichiers lourds, etc)



[https://en.wikipedia.org/wiki/Content_delivery_network]

Caches et ORM

- Cache mémoire intégré au framework
 - EntityManager en Java
- Cache de 2e niveau:
 - Au sein d'un même serveur: partagé entre les instances d'EntityManager
 - En Java: `@Cacheable` pour indiquer les entités pouvant être mises en cache
- Peut être utilisé dans des clusters, mais attention à la cohérence des caches entre machines

kv-stores pour le cache

- Systèmes clé-valeur très utilisés pour le cache applicatif
 - Pas besoin de requêter la valeur pour du cache
- Permet de partager un cache entre des instances de serveur hébergeant le code métier
- 2 gros acteurs: Memcached et Redis

Memcached

- clés et valeurs textuelles
- rapide
- distribué

Redis

- Clés pouvant être grosses (512 Mo, même si déconseillé)
- Valeurs complexes possibles: « It supports data structures such as strings, hashes, lists, sets, sorted sets with range queries, bitmaps, hyperloglogs and geospatial indexes with radius queries. »
- Persistance des données:
 - Utilisation possible comme cache d'écriture type write-back
 - Stockage de données de calcul intermédiaire:
Permet d'éviter de relancer complètement un calcul en cas de perte d'un noeud
- Distribué
- En général conseillé pour les nouveau projets

Caches distribués

- Deux vecteur de passage à l'échelle:
 - Réplication
 - Permet de répondre à une montée en charge sur une même ressource
 - Gestion de cohérence plus complexe/coûteuse
 - Distribution des données (sharding)
 - Basée sur la valeur de la clé (en général via hashage)
 - Problématique de partitionnement pour étaler correctement la charge

Caches distribués: mise en place du partitionnement

- Frontend qui distribue les requêtes sur les réplicats / partitions (*shards*)
 - Exemple: Twemproxy (memcached et Redis)
 - Possibilité de le faire fonctionner en cluster (évite de devenir un *single point of failure*)
- Interrogation d'un élément du cluster avec redirection vers le bon *shard* (Redis)
- Gestion des shards par le client:
les membres du cluster ne se connaissent pas entre eux

Importance des expérimentations

- Difficile de prévoir l'impact réel du cache
- Nécessité d'instrumenter les serveurs pour
 - vérifier les apports du cache
 - régler les paramètres de cache
- Mesure standard de l'utilité du cache:
hit ratio = #données trouvées / #demandes

Quelques références

- <http://infrastructure.smile.eu/Tout-savoir-sur/Principes-d-architecture-et-outils-open-source/Le-cache/Principes-du-cache>
- <https://www.nginx.com/blog/nginx-caching-guide/>
- <http://deptinfo.cnam.fr/new/spip.php?pdoc4647>
- <https://redis.io/documentation>
- https://fr.wikipedia.org/wiki/M%C3%A9moire_cache
- https://fr.wikipedia.org/wiki/Cache_web
- https://en.wikibooks.org/wiki/Java_Persistence/Caching