ApuntsTema1.pdf

onafolch

Desenvolupament d'Aplicacions de Dades Massives

3º Grado en Ingeniería de Datos

Escuela de Ingeniería
Universidad Autónoma de Barcelona

BIG DATA

STORING AND RETREIVING LARGE DATA VOLUMES

1. DATABASE

La base de dades més simple està formada per claus i valors (key, value). Si necessitem un valor, demanem la seva clau amb get(key).

- db_set(key, val): afegeix al final del fitxer (BD). Si sobreescric una clau, al fer el get em retornarà l'última.
- db_get(key): busca la paraula a la BD i es queda amb l'última.

Aquesta BD té un bon rendiment d'escriptura, perquè afegir al final del document és ràpid (log-file: seqüència de registres només per afegir). Però té un mal rendiment de lectura si la BD té molts registres, perquè cada consulta comença al primer element i llegeix cada element del fitxer des del principi fins al final (ja que es queda amb l'última instància de la clau que estem buscant). Per tant, té una complexitat de O(n), on n és el nombre de registres.

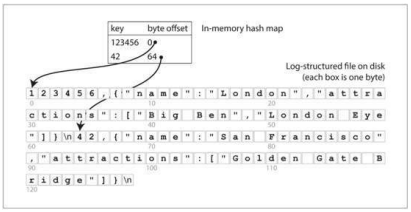
Exemples de complexitat: O(log n) - low cost, O(n*m) - high cost, O(n) - high cost, O(1) - low cost.

Per accelerar les lectures fem ús dels índex, la qual cosa ens permet cercar valors de manera eficient. Hem de generar i emmagatzemar metadades per trobar els valors ràpidament en una estructura de dades addicional derivada de les dades originals, de tal manera que no afecta les dades ni el contingut de la BD original. Però les actualitzacions dels índex afecten a les operacions d'escriptura de la BD, ja que cal afegir les dades i després actualitzar l'índex.

Tipus d'índex comuns: hash, sorted string tables, log-structures merge trees.

Hash

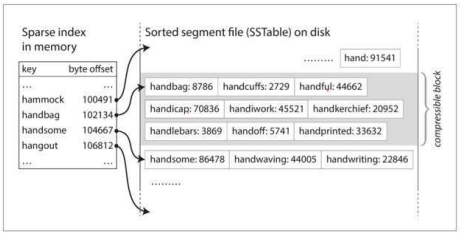
Mapa hash a la memòria que es guarda la posició de les claus en un fitxer (per guardar key-values). Cada element té dos valors, la clau i la posició. Al fer la lectura la capçalera del disc s'ha de col·locar a la ubicació de les dades (hora de cerca), i es llegeix la operació del valor del disc. Quan volem escriure en una mapa hash, afegim la clau i el valor al final del fitxer, i afegim una nova posició de clau al mapa.



Els principals problemes del mapa hash són que ha de cabre en memòria (com que el patró d'accés és aleatori pot haver col·lisions entre claus) i que no es pot aplicar en datasets amb moltes claus.

Sorted String Tables

Cada bloc té parells de clau-valor ordenats per clau. Cada bloc té només una instància de cada clau. Ara necessitem una nova operació: merge and sort (Si per exemple tenim 3 segments, agafem la clau més petita i la posem al fitxer de sortida i ho repetim). Per saber si existeix una clau no necessitem un índex de totes les claus a memòria, sinó que tindrem un índex per uns quants kilobytes, per tant, tindrem tantes claus com blocs.



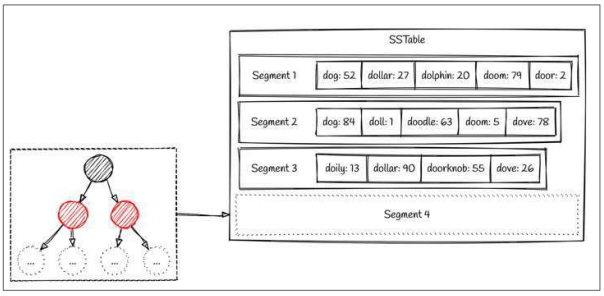
Per buscar una paraula, si la clau es troba a l'índex anirem a la posició del bloc, si no la trobem a l'índex, anirem al bloc anterior i buscarem la clau.

Log-structures merge trees (LSM Trees)

Per carregues de treball de moltes escriptures, on només s'accepten operacions d'escriptura seqüencials (amb això tindrem un bon rendiment d'escriptura). Estan darrere de moltes BD comercials (Google BigTable, AWS Cassandra, Scylla, RocksDB). Les dades estan al disc en SSTables, les quals estan formades per fitxers anomenats segments, que

són inmutables un cop s'han escrit al disc. A diferència del hash, tenim un bon rendiment per consultes d'interval cat0000-cat9999. Funciona bé quan el conjunt de dades és més gran a la memòria.

Quan s'escriuen nous valors (key, value), es guarden a la memtable (estructura de key-value ordenada a memòria). Quan aquesta està plena, s'escriu a disc com un nou segment de la SSTable. Per llegir les dades, començariem per llegir l'últim segment per trobar la clau, i si no la trobem, aniriem mirant els següents més recents, fins trobar-la. Això significaria que podríem retornar les claus que s'han afegit més recentment més ràpid. Una optimització senzilla és utilitzar el sparse índex a memòria (igual que a SST).



A mesura que passa el temps, aquest sistema anirà acumulant segments, els quals s'han de netejar i mantenir per tal d'evitar que hi hagi masses. Per aquesta raó compactarem les dades. Si tenim les mateixes claus en diferents segments, amb la compactació tindrem un únic segment amb la clau només una vegada.

Un problema que tenim és si el sistema falla, ja que totes les dades que estiguin a la memtable es perdran. Una solució seria tenim un altre log-file al disk sense un ordre aplicat, i quan el sistema falli, reconstruirem la memtable a partir d'aquest log-file afegint totes les claus i valors.

Limitacions:

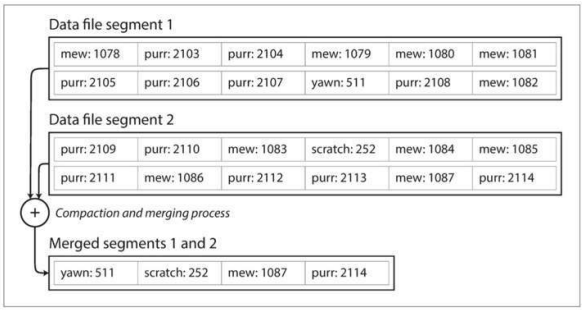
- La operació de compactació afecta al rendiment de lectura i escriptura, ja que les lectures s'han d'esperar a que acabi la compactació.
- S'ha de compartir la capacitat d'escriptura del disc amb tres operacions (logging, copiar la memtable a un segment, background compact).
- Write burst to memtable: la compactació no fa front a block merge requests, les lectures són més lentes perquè tenim més blocs sense compactar, i ens podem quedar sense espai al disc.

Espai al disc

Si ens quedem sense espai al disc, tenim tres opcions. La primera és trencar el fitxer de log en fitxers de mida fixada, que la següent operació s'escrigui a un altre fitxer, i finalment compactar els fitxers per estalviar espai.

Si tinc un fitxer amb claus i valors amb claus repetides, puc compactar les dades eliminant duplicats i guardant només l'últim clau-valor. En canvi, si tenim dos fitxers (dos blocs), podem generar nous blocs per substituir els antics, fent el mateix que hem dit abans (guardar només l'últim clau-valor entre els dos fitxers).

Cada segment té el seu mapa hash a memòria. Quan volem fer una cerca, busquem la clau al mapa hash més recent, si no està, al segon més recent, etc , fins trobar el valor. Hauriem de tenir un nombre limitat de segments per evitar operacions addicionals de lectura hash.



Té avantatges utilitzar un fitxer que només es pot afegir al final (immutable). Afegir i combinar són operacions seqüencials d'escriptura (és més ràpid que les operacions d'escriptura aleatòries, en qualsevol posició), a recuperació d'errors és senzilla d'implementar i les operacions de fusió de blocs gestionen els problemes de fragmentació a temps.

Casos d'arquitectura key-value

- KingDB: log-structured storage
- LevelDB: LSM-Trees