ESTRUCTURES D'ACCÉS

Una de les visions més simples del disseny físic d'una base de dades podria ser veure-hi només taules i que cada taula estigui emmagatzemada en un fitxer del sistema operatiu. Amb aquest disseny tots els accessos per fer INSERTs, DELETEs, UPDATEs o SELECTs requeririen, en general, llegir la totalitat o gran part dels fitxers que, eventualment, podrien ser voluminosos de manera que aquestes lectures podrien requerir molt de temps fent que el rendiment de la base de dades fos molt pobre. Una manera de mesurar aquest rendiment és comptar el nombre de blocs de disc que cal accedir per executar una sentència. I l'objectiu principal que ens hem de marcar és minimitzar aquesta mesura.

Uns dels elements addicionals a les taules que trobem en una base de dades són estructures d'accés a les dades que ens permeten disminuir el nombre de blocs accedits, organitzant les tuples convenientment i construint índexos que ens dirigeixen de manera més directa als blocs on hi ha les dades que cerquem, en comptes de llegir tot o gran part dels fitxers, aprofitant la possibilitat d'accedir directament a un bloc d'un fitxer sense necessitar de llegir seqüencialment tots els anteriors.

Per simplificar, de moment, diguem que un índex el definim sobre un atribut d'una taula per poder trobar amb pocs accessos files de la taula en funció del valor que tenen en l'atribut indexat. Un índex és una col.lecció d'*entrades*, organitzades de manera diferent en funció del tipus d'índex. Una entrada és una parella <valor, informació>, un valor possible de l'atribut indexat i una informació que permet trobar les files que contenen aquell valor.

A la transparència 4 hi ha moltes combinacions possibles d'organitzar una taula, mantenint-la o no ordenada, indexada o no i, en cas afirmatiu, amb un índex B+ o hash. De totes aquestes, analitzarem en aquesta unitat les més habituals en els SGBD comercials:

- taula no ordenada i sense índex
- taula no ordenada i amb un índex B+ on cada entrada té l'adreça d'una fila (de manera que si la taula té repetits en l'atribut indexat hi haurà diverses entrades amb el mateix valor)
- taula ordenada per l'atribut indexat i amb un índex B+ on cada entrada té l'adreça d'una fila (de manera que si la taula té repetits en l'atribut indexat hi haurà diverses entrades amb el mateix valor). D'aquesta combinació en direm índex cluster.
- taula no ordenada i amb un índex hash on cada entrada té l'adreça d'una fila (de manera que si la taula té repetits en l'atribut indexat hi haurà diverses entrades amb el mateix valor).

A la transparència 5 hi teniu un recordatori de la disposició habitual de files en els blocs d'un fitxer.

A la 6, un exemple d'una taula indexada amb un B+. En un B+ cada node (que físicament es correspon amb un bloc de disc) té uns quants dels valors de l'atribut indexat que apareixen a la taula indexada. En el subarbre que hi ha a l'esquerra d'un valor només hi ha valors menors, mentre que en el subarbre que hi ha a la dreta hi ha valors majors o iguals. Aquest exemple és molt lluny de la realitat, normalment en un node hi haurà desenes o algun centenar de valors i el nombre de files de la taula serà de milers o algun milió. Les fletxes de la figura són encadenaments que efectivament es guarden per poder llegir el fill d'un node o el germà d'una fulla amb un sol accés. Les fulles són les que contenen les entrades de l'índex. Aquesta organització permet trobar una entrada baixant des de l'arrel cap a les fulles, decidint a cada nivell per quin fill cal baixar comparant el valor que busquem amb els que conté el node d'aquell nivell. El nombre d'acessos correspon, doncs, a la longitud del camí que va de l'arrel a la fulla. Definim l'ordre de l'arbre (que denotarem amb la lletra d) com la meitat de les entrades que caben en una fulla, que equival al nombre de valors que caben a cada node. Els nodes, però, procurarem que no estiguin plens sinó carregats només fins a un determinada proporció, que anomenem factor de càrrega. Per defecte, suposarem que el factor de càrrega és 2/3.

A la transparència 7 hi ha un exemple d'índex cluster. L'arbre funciona igual que en l'exemple anterior, és un B+ igualment. La diferència és que ara la taula està ordenada, per això les fletxes que van de les fulles a la taula no es creuen entre elles (les entrades a les fulles també estan ordenades). Com que hem de mantenir la taula ordenada, els blocs del fitxer que guarda les files també els deixem amb un certa proporció d'espai lliure, per poder fer insercions al mig de la taula sense haver de desplaçar files d'un bloc a un altre.

A la transparència 8 hi teniu un exemple d'índex hash senzill i molt optimista, que és el model que farem servir en aquesta assignatura. En un hash les entrades es distribueixen en buckets segons el valor d'una funció, la funció de hash. Aquesta funció, donat un valor de l'atribut indexat, retorna un nombre que identifica a quin bucket li toca anar aquest valor. Evidentment, pot passar que un bucket s'ompli i una nova entrada que li toca anar a aquell bucket no hi càpiga (d'aquesta entrada se'n diu excedent), però nosaltres suposarem que això no passarà.

Presentades les opcions d'indexació, ara passarem a analitzar quan espai ocupa cadascuna i quin cost tenen les operacions d'un repertori representatiu en cada opció. Aquests costos els expressarem en funció de les variables que es presenten a la transparència 9.

Abans de començar, però, la transparència 10 ilustra amb un exemple com determinar l'alçària d'un arbre en funció del nombre d'entrades - que coincideix amb el nombre de files de la taula, |T| -, l'ordre de l'arbre (d) i el factor de càrrega (l). El primer que fem és

calcular, donat l'ordre de l'arbre i el factor de càrrega, quantes entrades tindrem a cada node (u):

$$u = 2 * d * l$$

A partir d'aquí, podem calcular el nombre de fulles. que és |T| / u. I, sabent que cada node té *u* fills, sabem que cada vegada que pugem un nivell es dividirà el nombre de nodes per *u*. Per això el nombre de nivells serà $\log_u(|T|)$ i l'alçària (h) que és la longitud del camí que va de l'arrel a una fulla, és el nombre de nivells menys 1:

$$h = \log_u(|T|) - 1$$

A la transparència 11 hi teniu l'espai que ocupa cada opció:

- La taula sola ocupa, per definició, B blocs.
- La taula amb B+ ocupa la suma dels blocs que ocupa la taula i els blocs ocupats per l'arbre (que és el sumatori, per cada nivell de l'arbre, dels nodes que té el nivell)
- La taula amb cluster ocupa la suma dels blocs que ocupa la taula i els blocs ocupats per l'arbre. Però recordem que la taula ara té els blocs ocupats a 2/3 de la seva capacitat. Per tant, si la taula amb els blocs plens ocupa B, ara la taula ocupa B' tal que B'*2/3 = B. O sigui, B' = 1.5 * B.
- La taula amb hash ocupa la suma dels blocs que ocupa la taula i els blocs ocupats pels buckets més un bloc destinat a la gestió del hash. Pel que fa al nombre de buckets, el podem calcular de la mateixa manera que abans hem calculat el nombre de fulles de l'arbre, però amb un factor de càrrega de 0.8. És a dir, en un bucket hi tindrem 2 * 0.8 * d entrades i, per tant, necessitem |T| / (2 * 0.8 * d) = |T| / (2 * 4/5 * d) = 5/4 * |T| / 2d = 1.25|T|/2d.

Passem ara a analitzar el cost en accessos a disc de diferents operacions en funció de l'opció amb què s'hagin organitzat les dades. A la transparència 12 hi teniu el repertori d'operacions i a les posteriors els costos:

- El table scan sempre el podem fer ignorant l'existència d'un índex si volem, però a la transparència s'hi ha posat el cost suposant que es fa el scan a través de l'índex. En particular
    - En el cas de B+, suposem que recorrem les fulles de l'arbre i, per a cada entrada, accedim a la tupla referenciada.
    - En el cas de hash, suposem que recorrem els buckets i, per a cada entrada, accedim a la tupla referenciada.
- En la cerca d'una tupla:
    - en una taula sense índex suposem que en mitjana haurem de llegir seqüencialment la meitat de la taula.
    - suposem que l'arrel dels arbres ja la tenim a memòria, per tant llegir la fulla on hi ha l'entrada que cerquem costa *h* accessos i en sumem un per accedir a la fila de la taula.

    - amb hash només hem de calcular la funció de hash, llegir el bucket que ens indiqui la funció i llegir el bloc de la taula que ens indiqui l'entrada trobada. Fixeu-vos que amb hash (el model optimista que hem adoptat) el cost és constant mentre que en les altres opcions creix amb la mida de la taula.
- En la cerca de diverses tuples:
    - en una taula sense índex hem d'arribar fins al final perquè no sabem quantes tuples hem de trobar.
    - amb un B+ farem *h* accessos per trobar la primera fulla que té una entrada que ens interessa i després anirem llegint entrades (i fulles per la dreta si cal) mentre trobem valors dins del rang que busquem (o repeticions del valor que busquem). Si suposem que hem de trobar |O| entrades (ja veurem en sessions posteriors com podem estimar aquest valor) i tenint en compte que ja n'hem trobat 1, ens en falten |O| - 1, que estaran distribuides en (|O| - 1) / u fulles més. Un cop recuperades les |O| entrades, farem un accés a la taula amb cada una.
    - amb un cluster, que té la taula ordenada, accedirem a la primera fila de la taula que ens interessa (això ens costarà h + 1 accessos) i després anirem llegint files (i blocs si cal) mentre trobem valors dins del rang que busquem (o repeticions del valor que busquem). Si suposem que hem de trobar |O| files (ja veurem en sessions posteriors com podem estimar aquest valor) i tenint en compte que ja n'hem trobat 1, ens en falten |O| - 1, que estarien distribuides en (|O| - 1) / R blocs més. Però com que els blocs estan ocupats en 2/3 de la seva capacitat, hem de multiplicar per 3/2.
    - amb un hash llegirem un bucket per cada valor (totes les repeticions d'un valor estan al mateix bucket segons la suposició que hem fet) i farem un accés a la taula per cada fila que cal recuperar. Fixem-nos que en el cas del hash només podem cercar rangs que es puguin expressar com

      `SELECT v FROM T WHERE v IN (`v_1`, `v_2`, ...`v_n`)`
- A les insercions i supressions, hem de tenir en compte que per modificar un bloc cal llegir-lo, modificar-lo a memòria i tornar-lo a escriure.