Appunti MAADB - Federico Torrielli

Capitolo 8: Storage e Indexing

Tipi di file structures

- **Heap file**: file di tipo non ordinato, i record nei file di heap sono archiviati in modo random tra le pagine del file
- **Index file**: struttura dati che organizza i data record in modo da ottimizzare alcune operazioni di ricerca. Un index ci permette innanzitutto di recuperare in maniera efficiente tutti i record che soddisfano le condizioni di ricerca specificate dalla *search key* nell'index. I record che fanno parte dell'index file vengono detti **data entry**

Esistono tre diverse alternative per che cosa archiviare come una data entry nell'indice:

- Data entry come data record
- Data entry come una coppia *<k, rid>*
- Data entry come una lista di coppie *<k,rid>*

Tipi di indici

Clustered/Unclustered

- Clustered: dati vengono ordinati come le data entry in un indice, ovvero seguendo l'ordine della search key
- Unclustered: altrimenti

Primary/Secondary

- Primary: indice definito su un set di campi che include la primary key
- Secondary: altrimenti

Strutture dati per indici

Indici basati su hash I record in un file sono organizzati in **buckets** dove il bucket rappresenta una catena di pagine (primary page –> secondary pages). Per sapere dove risiede il bucket di una tupla specifica viene applicata una funzione di hash h alla chiave di ricerca. Questa operazione viene portata a termine in una o due I/O.

Indici basati su alberi Al livello più basso troviamo le foglie, dove risiedono le data entries, mentre i nodi servono principalmente per guidare la ricerca in tempo logaritmico. La testa è la root.

Il numero di I/O in questo caso è equivalente al traversamento dell'albero dalla testa alla foglia interessata + la ricerca se abbiamo pagine di overflow.

Strutture dati come il B+Tree aiutano in questo caso dato che mantengono l'albero sempre bilanciato.

- Fan-out: il numero medio di figli di un nodo interno

Confronto tra strutture files

Il nostro modello di costi è il seguente:

- B: numero di data pages quando i record occupano pagine senza spazio extra
- R: numero di record per pagina
- D: tempo medio di scrittura o lettura di una pagina
- C: tempo medio per processare un record
- H: tempo di hashing

  Nota: al tempo della scrittura del libro (il lontano 2004), D=15ms e C,H=100ns

File di Heap

- Scan: $B(D + RC)$ dato che dobbiamo recuperare ognuna delle B pagine mettendoci D tempo per pagina, e processare R record in tempo C per record
- Ricerca per uguaglianza: $0.5B(D + RC)$ dato che mediamente facciamo la scansione di metà della pagina, e applicare il ragionamento precedente
- Ricerca per range: stesso costo dello scan
- Inserzione: $2D + C$ assumendo che i record vengono sempre messi in append, leggiamo l'ultima pagina nel file, aggiungiamo il record e facciamo una write
- Cancellazione: $C + D$ stesso ragionamento del precedente.

File sorted

- Scan: vedi lo scan del file di Heap
- Ricerca per uguaglianza: $Dlog_2(B) + Clog_2(R)$ se prendiamo per dato il fatto che le pagine siano archiviate sequenzialmente nella struttura sorted allora la ricerca si riduce ad una ricerca binaria.
- Ricerca per range: stesso costo della ricerca per uguaglianza.
- Inserimento: Costo della ricerca per uguaglianza + $B(D + RC)$ per ovvie ragioni, ovvero si tratta di un'operazione costosa
- Cancellazione: stesso ragionamento del precedente

File clustered

Nota: i file clustered vengono considerati con una occupancy del 67%

- Scan: $1.5B(D + RC)$ dato che tutte le pagine devono essere esaminate, praticamente come i sorted files, ma contando l'occupancy
- Ricerca per uguaglianza: $Dlog_f(1.5B) + Clog_2(R)$ ovvero il costo del traversamento dell'albero fino alla pagina desiderata + il costo della ricerca

binaria della pagina. In ogni caso si tratta di un miglioramento rispetto ai file sorted.
- Ricerca per range: stesso ragionamento del precedente
- Inserimento: $Dlog_f(1.5B) + Clog_2(R) + D$ ovvero il costo della ricerca + il costo di una write
- Cancellazione: stesso ragionamento del precedente

Heap files con Unclustered tree index

- Scan: facciamo una scansione del livello delle foglie dell'indice e per ogni data entry andiamo a prendere il corrispondente data record dal file selezionato ottenendo quindi data record nell'ordine di sort specificato, il costo dunque sarebbe $BD(R + 0.15)$, ovvero in termini di I/O molto costoso.
- Ricerca per uguaglianza: il costo dell'utilizzo di un unclustered index dipende fortemente dal numero di record che qualificano la ricerca. Mediamente il costo sarebbe $D(1 + log_f(0.15B))$
- Ricerca per range: D($log_f(0.15B)$+ #record che matchano)
- Inserimento: prima inseriamo il record nell'heap file, a costo $2D + C$, e addizionalmente la corrispondente data entry nell'indice. Trovare la giusta foglia costa tempo lineare, per poi andare a fare le write a costo 2D. Il costo totale sarebbe di $D(3 + log_F(0.15B))$
- Cancellazione: Ricerca + $2D$ per ovvi motivi

Heap files con Unclustered Hash index

> Nota: assumiamo che ogni data entry sia 1/10 della grandezza del data record e che non ci siano overflow chains, consideriamo inoltre una occupancy dell'80%.

- Scan: mentre le data entry possono essere prese ad un costo minimo, è invece costosissimo prendere ogni data record, al costo di $BR(D + C)$
- Ricerca per uguaglianza: Il costo di identificare la pagina della data entry è `H`, se il bucket ha una pagina sola allora il costo si riduce a `D`. Il resto della scansione è un altro `D`. Il costo totale allora si porta a `2D`.
- Ricerca per range: si applicano tutte le problematiche varie dell'hashing sulla ricerca per range. Dobbiamo fare scan di tutti i record al costo di $B(D + RC)$.
- Inserimento: `4D` –> `2D` per l'inserimento nell'heapfile e `2D` per la scrittura della data entry.
- Cancellazione: Ricerca + $2D$ per ovvi motivi.

Tabella dei costi (solo IO)

Chiavi di ricerca "composite" o concatenate

Una chiave di ricerca è composite o concatenata quando contiene più valori per un indice.

Cost of Operations

	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap	BD	0.5BD	BD	2D	Search +D
(2) Sorted	BD	Dlog 2B	D(log 2 B + # pgs with match recs)	Search + BD	Search +BD
(3) Clustered	1.5BD	Dlog F 1.5B	D(log F 1.5B + # pgs w. match recs)	Search + D	Search +D
(4) Unclust. Tree index	BD(R+0.15)	D(1 + log F 0.15B)	D(log F 0.15B + # pgs w. match recs)	Search + 2D	Search + 2D
(5) Unclust. Hash index	BD(R+0.125)	2D	BD	Search + 2D	Search + 2D

☛ *Several assumptions underlie these (rough) estimates!*

Figure 1: Tabella dei costi, tenendo conto dei soli IO

Se la chiave di ricerca è composite una query di uguaglianza è tale da ogni valore nella chiave di ricerca associata ad una costante. Ad es: *<age = 20, sal = 200>*. Nel caso delle ricerche per range non tutti i valori possono essere associati ad una costante, vd. *20, sal < 150>*.

Una chiave di ricerca composite supporta molte operazioni comode di ricerca che possono essere effettuate velocemente su strutture ad albero. Dall'altro lato, invece, un indice composite deve essere aggiornato ogniqualvolta ci sia un'operazione (insert, delete, update) che modifichi *qualsiasi* valore nella chiave di ricerca. Nel caso in cui si stia usando un B+Tree avremmo il problema dell'aumento dei livelli, anche se la compressione potrebbe certamente aiutare in questo caso.

Capitolo 9: Dischi e files

La gerarchia di memoria

Primary Storage (CPU cache && RAM) > Secondary Storage (Dischi magnetici && SSD) > Tertiary Storage (Nastri e altre diavolerie ancora + lente)

Dischi magnetici

- Costo di IO per un disco: Seek Time + Rotational Delay + Transfer Time

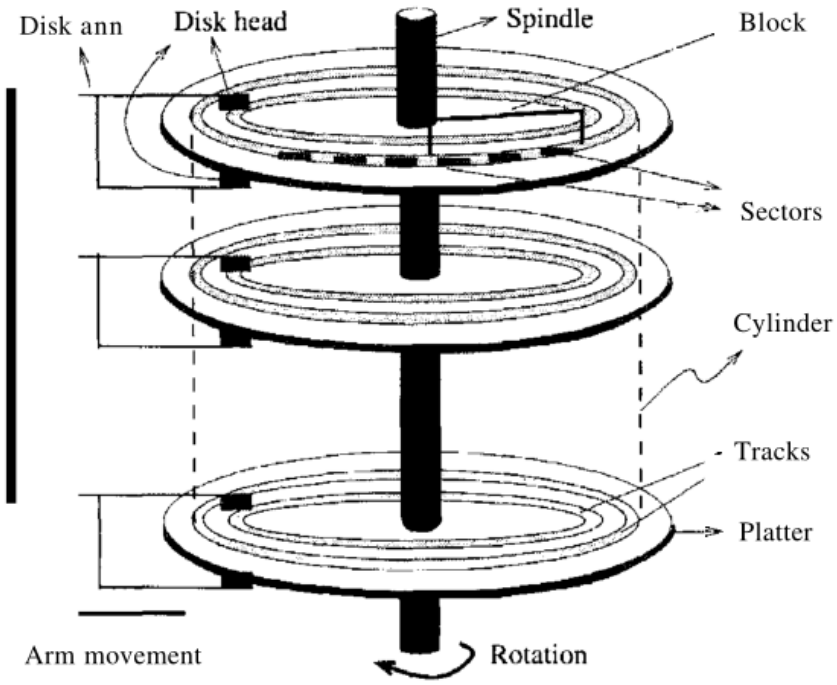

Figure 2: Struttura di un disco

Buffer manager: introduzione

Definizione: il buffer manager è il layer sw responsabile di portare le pagine dalla memoria secondaria alla memoria primaria. Esso partiziona logicamente la memoria primaria in una collezione di pagine, chiamata la *buffer pool*. Le pagine nella buffer pool sono chiamate *frames*.

- *pin-count*: il numero di volte che un frame è stato chiamato ma non rilasciato. Se è 0, la pagina è la favorita per il rimpiazzamento.
- *dirty*: il valore booleano che indica se la pagina è stata modificata da quando è stata portata nella buffer pool.
- *pinning*: l'azione pin-count++, contrario di unpinning.

Buffer replacement policies

1. **LRU**: sceglie per il replacement la pagina utilizzata da meno tempo. Può essere implementato con una *coda di puntatori ai frames con pin_count == 0*, un frame diventa candidato quando è aggiunto alla tail della coda (ovvero ha pin count zero) e viene sempre preso il primo della head per il replacement.
2. **Clock**: creato per emulare e migliorare LRU (dunque soffre degli stessi problemi e ha gli stessi vantaggi, anzi, meno overhead). L'idea è quella di scegliere la pagina utilizzando una variabile *current* (ovvero, la lancetta dell'orologio), che scorre circolarmente dal frame 1 a N. Ogni frame ha un bit *referenced* che ci dice quando *pin_count == 0*. Il frame current è quello selezionato per essere analizzato, se è rimpiazzabile (ovvero quando pincount == 0 e referenced == false) allora viene rimpiazzato, altrimenti si va avanti di uno, come nell'orologio. Nell'analisi, se il referenced è messo true, allora l'algoritmo lo metterà a false ed andrà avanti.

Problema di LRU/Clock –> **sequential flooding**: se diversi user richiedono scan sequenziali risultiamo nella lettura di OGNI singola pagina del file, nessuna esclusa. In questi casi forse la strategia migliore è la MRU, che non soffre di questo problema.

Prefetching di pagine e forcing

Per predirre dei pattern di reference molto spesso si ricorre al prefetching delle pagine. Il buffer manager anticipa le richieste di diverse pagine (se ha tempo) e fa fetch delle pagine che serviranno *prima* che siano effettivamente richieste (un po' come prevedere il futuro, dai...). Soprattutto se il prefetching è fatto su pagine che di loro natura non sono contigue, questo può nettamente velocizzare l'IO una volta che verrà fatta la richiesta.

Un DBMS può anche richiedere una *force* della pagina sul disco, ovvero quando la pagina in memoria è copiata su disco volutamente.

Separazione dei domini (LRU)

Approccio di buffer replacement che invece che trattare tutti i frame di buffer in modo equivalente, trattiamo porzioni di buffer e file diversi con metodi di rimpiazzamento specifici a quel tipo di file o porzione di buffer. Pagine vengono classificate in tipi, ognuno dei quali è separatamente gestito nel suo dominio associato di buffer dalla politica LRU.

Questo aggiunge un overhead non indifferente se gestito male, o se la separazione viene effettuata senza regole ben precise.

Algoritmo del working-set (MRU)

Prevede che la priority di una pagina debba essere determinata in base alla frequenza rispetto a cui viene utilizzata la relazione sotto cui la pagina fa parte. Si associa dunque una frequenza ad un *gruppo di pagine* che rappresenta la relazione considerata, invece che utilizzare LRU per tutte le pagine di ogni relazione. Dato che consideriamo un solo gruppo alla volta, riduciamo anche il carico I/O dato dal fatto che non riordiniamo più la coda come in LRU.

Hot-Set (LRU)

Un set di pagine su cui si verifica un comportamento *looping* è detto Hot-Set. Ci basiamo sull'idea di *Hot Point*, ovvero il più piccolo numero di frame che si può allocare alla query garantendo un drastico calo di page fault che ottengo rispetto ai precedenti. Il numero dei buffer è *per query*, gestite con politica LRU, ed il numero di buffer è appunto predetto dall'Hot-Set.

Nel caso peggiore stimiamo un numero eccessivo di frame, dato che non possiamo prevedere il futuro, presentando quindi uno spreco.

Query Locality Set Model

Dato che i DBMS supportano un range ristretto di operazioni, possiamo osservare pattern di riferimento alle pagine create da operazioni in modo costante. Le operazioni che fa il database possono essere perciò scomposte in un 9 pattern di riferimento:

- **Straight Sequential**: accesso banalmente sequenziale. Dato che leggiamo una pagina e poi la sostituiamo per sempre, ci serve solo un frame di buffer
- **Clustered Sequential**: record in un *cluster set* (sono record con lo stesso valore sulla chiave), dovrebbero essere mantenuti in memoria allo stesso tempo, se possibile. In questo caso, le dimensioni del cluster determinano il numero di frame di buffer.
- **Looping Sequential**: praticamente emula MRU. Utilizzato nel join naive.
- **Independent Random**: simile a straight sequential ma le pagine vengono selezionate randomicamente in modo non-clustered. Utilizzato nell'accesso ai file con indice unclustered. Anche qui possiamo utilizzare 1 frame solo.

- **Clustered Random**: vale lo stesso discorso ma per cluster set.

Ed ora analizziamo i pattern di riferimento in cui vengono utilizzati indici:

- **Straight Hierarchical**: percorro sequenzialmente in cammino dell'albero che porta dalla radice alla foglia interessata. Simile a SS. Mi basta 1 frame.
- **Hierarchical + SS**: l'indice viene attraversato una sola volta, seguito da uno scan attraverso le foglie, similare a SS, basta 1 frame. Utilizzato nelle index based queries.
- **Hierarchical + CS**: discorso del precedente ma con cluster set.
- **Looping Hierarchical**: le pagine più vicine alla radice sono solite avere più accessi. La probabilità di accesso all'i-esimo livello è inversamente proporzionale all'i-esima potenza del fattore di fan-out. Ci conviene mantenere pagine in un livello alto di memoria preferenzialmente.

DBMIN

Algoritmo che utilizza il QLSM e che alloca i buffer e li gestisce *per file*: una *file instance* in questo caso è tutto quel range di file che vengono aperti da una query, e di conseguenza anche le sue pagine. A ciascun file vi è associato un pattern di accesso buono di QLSM.

In generale:

- Se il buffer contiene una pagina che non appartiene a nessun locality set, viene messa nella lista globale delle pagine libere.
- Pagine nel buffer non possono appartenere a più di un locality set. Un'istanza di file è considerata *owner* di tutte le pagine nel suo locality set.

L'algoritmo funziona nel modo seguente: quando una pagina viene richiesta da una query, facciamo una ricerca nella tabella globale, seguita da un aggiustamento al suo locality set, ci sono tre casi.

1. Pagina trovata nella tabella gloale e nel locality set: leggiamo la pagina, modifichiamo solo le statistiche
2. Pagina trovata nella tabella globale ma non nel locality set: se la pagina ha un owner, allora restituiamo la pagina alla query che la richiede. Altrimenti, la aggiungiamo ad un locality set. Se questo eccede il numero di buffer massimi allocati per un file, la dobbiamo rilasciare con la politica di replacement corrente.
3. La pagina non è in memoria: piangiamo tantissimo, per poi portarla in memoria e procedere con il passo 2.

Index Based Query

Tipo di query che si può valutare interamente senza dover accedere al file di dati. La risposta avviene utilizzando solo l'indice. Lo possiamo fare solo in file

non-clustered, dato che a livello di foglia abbiamo tutte le chiavi indicizzate, altrimenti non avremmo abbastanza informazioni.

Le index based query vengono utilizzate pesantemente nell'ambito della query evaluation per ridurre i tempi di query al minimo.

Heap Files: implementazione

Linked List di pagine La prima alternativa consiste nel mantenere una doppia linked list di pagine.

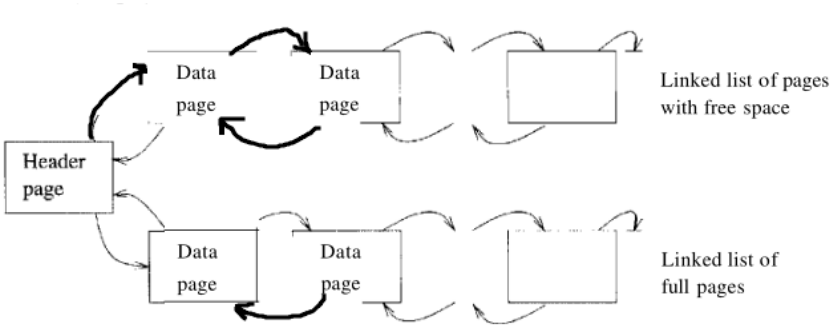

Figure 3: Linked list

- La prima pagina è sempre conosciuta dal DBMS perchè abbiamo una tabella che contiene tuple <heap_file_name, page_1_address>, e la prima pagina che collega entrambe le linked lists è detta la **Header page**
- Nel caso di cancellazione, la singola pagina viene rimossa dalla lista e il disk space manager la dealloca.
- Un grosso svantaggio sta nel fatto che virtualmente tutte le pagine nel file saranno nella free list se i record sono di lunghezza variabile, dato che ogni pagina sicuramente avrà qualche byte libero sempre! Per inserire un record dobbiamo infatti esaminare tante record pages nella free list prima di trovarne una con abbastanza spazio libero, ed è un lavoro di complessità lineare non banale.

Directory di pagine La seconda alternativa consiste nel mantenere una directory di pagine, che è poi implementata come una linked list di directory di pagine.

- Ogni directory entry identifica una pagina nell'heapfile.
- Lo spazio libero è organizzato mantentendo un bit per entry che indica quando la pagina corrispondente ha spazio libero o meno oppure con un counter che indica quanto spazio libero è effettivamente disponibile.

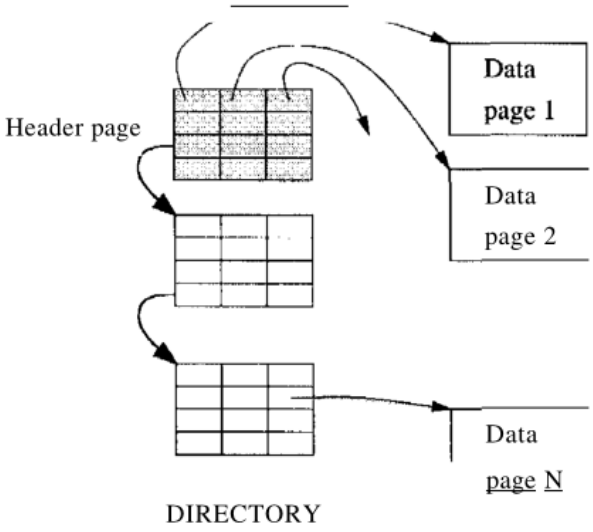

Figure 4: Directory di pagine

Formati della pagina

Possiamo pensare ad una pagina come una collezione logica di slot, ognuno il quale contiene un record.

RID (record id): <page_id, slot_number>

Fixed-Length Records Se tutti i record nella pagina sono garantiti essere *della stessa lunghezza fissata* allora gli slot dei record possono essere organizzati di conseguenza.

Vantaggi:

- Non occupa tanta memoria dato che manteniamo un'occupancy del 100%
- Quando un record viene eliminato, non dobbiamo cercare un modo per riempire lo spazio incompleto lasciato da quel record

Svantaggi: Va bene solo quando abbiamo record della stessa lunghezza, dunque non abbiamo flessibilità

Come teniamo traccia degli *slot liberi*?

1. Memorizziamo i record nei primi N slot e quando un record viene cancellato muoviamo l'ultimo record presente nel buco formatosi. In questo modo riusciamo a localizzare l'i-esimo record semplicemente con un calcolo dell'offset.

2. Organizziamo le cancellazioni grazie ad un array di bit, uno per slot. Quando un record viene cancellato, il suo bit viene messo a 0, uno altrimenti. Per trovare un record vuoto mi basterà cercare nell'array di bit in modo lineare.

Variable-Length Records Se i record sono di lunghezza variabile non possiamo organzzarli come nel caso precedente. Sorge dunque il problema di trovare lo spazio corretto in modo da avere il massimo riempimento di uno slot tutte le volte che facciamo un'inserimento e contemporaneamente tenere ben organizzati gli spazi vuoti in modo da non ritrovarsi con un'occupancy troppo bassa ma una memoria piena.

Vantaggi: Flessibile, va bene quando i record sono diversi in grandezza

Svantaggi: Il contrario dei vantaggi del Fixed-Length

All'inserzione: dobbiamo allocare la giusta quantità di spazio per esso. Non possiamo certamente scegliere uno slot più piccolo del record ma non dobbiamo sceglere neanche uno slot che lasci uno spazio troppo piccolo per un altro futuro record da mettere. Dobbiamo anche garantire in qualche modo che lo spazio libero sia tutto contiguo, in modo da trovarlo senza troppe ricerche tra tutti gli slot da controllare.

Directory di slot: il modo più flessibile di organizzare record a lunghezza variabile è mantenere una directory di slot per ogni pagina. Per ogni slot dobbiamo mantenere una coppia <*record_offset*,*record_length*>. L'offset del record mantiene il puntatore al record, e quando cancelliamo il record basta metterlo a -1. Il record può muoversi all'interno della pagina grazie al fatto che il **rid** non cambi quando il record viene mosso, ma cambia solo il record offset.

Gestire la memoria libera: per gestire gli slot liberi abbiamo bisogno di un puntatore alla zona degli slot liberi. Ma per averli liberi ci dobbiamo prima assicurare che dopo una reorganizzazione programmata dello spazio tutti questi slot siano contigui, seguiti dallo spazio libero.

Alla cancellazione: L'unico modo per rimuovere uno slot dalla slot directory è rimuovere l'ultimo slot se il record che lo punta viene cancellato.

Formati dei record

Mentre abbiamo discusso su come organizzare i record nelle pagine, ora passiamo a discutere su come organizzare i campi rispetto ad un record, ricordando che Pagina > Record > Campi (fields of a record).

NB: tutte le informazioni meta su come organizzare record e pagine fanno parte del **system catalog**

Fixed-Length Records I campi di un record di questo tipo vengono memorizzati in maniera contigua, e dato l'indirizzo di un record, l'indirizzo dell'i-esimo

field del record viene calcolato con l'offset

Variable-Length Records

1. Possiamo organizzare in maniera contigua i fields, separandoli con delimitatori (caratteri speciali come #), questo tipo di organizzazione richiede una scansione del record per trovare il field che ci interessa
2. Possiamo riservare dello spazio all'inizio del record per un'array di interi di offset, ogni intero significa lo starting address dell'i-esimo valore field del record. Grazie a questo metodo, molto efficace, possiamo avere accesso praticamente diretto al field del record. Otteniamo inoltre un altro vantaggio: non abbiamo problemi con i valori **null**. Quando dobbiamo memorizzare un valore null, semplicemente lo facciamo puntare al puntatore dell'inizio del field, e non occupa spazio.

Problemi:

- Modificare un field potrebbe causare l'ingrandimento di un record, che ci farebbe fare uno shift dei field che susseguono per fare spazio alle modifiche.
- Se modifichiamo un record questo non potrebbe più starci nello spazio della pagina ed essere spostato. In questo caso abbiamo un puntatore che punta ancora al record vecchio ma spostato.
- Un record potrebbe crescere talmente tanto che non ci sta in *nessuna pagina*, un questo caso dobbiamo spezzettare il record in mini-record.

Capitolo 10: Tree-Structured Indexing

Concetti generali

- Le strutture ad albero sono ottimali per le ricerche a range, molto buone per le inserzioni e le cancellazioni.
- Per quanto riguarda le equality selection non sono particolarmente efficienti come le strutture basate su hash ma permettono comunque la funzione.
- ISAM: albero statico, fatto per dati che non vengono frequentemente aggiornati
- B+Tree: struttura dinamica che si aggiusta molto bene ai cambiamenti e tenta sempre di tenersi bilanciato.
- Index entry: tupla <key, pointer>

ISAM: Indexed Sequential Access Method

- I dati stanno al livello delle foglie ed utilizziamo catene di overflow
- Struttura completamente statica, tranne per le catene di overflow (sperabilmente le minori possibili).
- Un nodo == una pagina di disco
- Quando il file viene creato, tutte le pagine foglia sono allocate sequenzialmente e ordinate sul valore della chiave di ricerca. Dopo vengono allocate le pagine del livello non-foglia

- Le inserzioni e le cancellazioni modificano solo il livello foglia, dato che la struttura rimane statica. Di conseguenza a seguito di una grande sequenza di inserzioni ci troviamo con lunghe catene di pagine di overflow. Per alleviare il problema teniamo una occupancy dell'80% a creazione dell'albero.
- ISAM aiuta con l'accesso concorrente: quando facciamo accesso ad una pagina, e ci mettiamo un lock, occupiamo tempo prezioso. Sull'albero ISAM ci basta bloccare il livello foglia, dato che sappiamo che gli altri livelli non verranno mai modificati.

B+Trees

- Albero bilanciato nel quale i nodi interni dirigono la ricerca e i nodi foglia contengono i dati
- Visto che la struttura viene modificata spesso, non allochiamo sequenzialmente le pagine foglia come in ISAM, ma le allochiamo come tutte, per poi far si che ogni foglia abbia un puntatore alla foglia successiva, e viceversa (doppio puntatore, avanti ed indietro).
- Le operazioni di inserzione e cancellazione tengono l'albero bilanciato
- Occupancy minima del 50% per ogni nodo (tranne la radice) se utilizziamo un algoritmo di cancellazione. Altrimenti semplicemente rimuoviamo il nodo ed aggiustiamo l'albero
- La ricerca ha il costo del traversamento dalla radice al nodo cercato, ovvero di lunghezza al massimo equivalente alla altezza dell'albero (per quello dobbiamo tenerlo bilanciato!!)
- B+Trees solitamente mantengono una occupancy del 67% in generale

Gestione dei duplicati nei B+Trees

1. Possiamo utilizzare banalmente delle pagine di overflow per i duplicati a livello di foglia
2. Possiamo trattarli come qualsiasi altro valore ed inserirli/cancellarli dall'albero. Il problema qui sorge quando cerchiamo di cancellarli e le data entry sono del tipo <key, rid>. In questo caso abbiamo tanti valori da cercare con la stessa *key*, rendendo la ricerca perciò inefficiente. Possiamo risolvere il problema usando *anche* il **rid** invece che solo la key come search key, trasformando l'indice in un indice univoco a tutti gli effetti.

Key compression L'altezza dell'albero dipende da:

1. Il numero di data entry
2. La grandezza delle index entry (che determina anche il fan-out dell'albero)

Visto che l'altezza dell'albero è log_fanout(#data_entry) e facciamo un'I/O per ogni salto d'altezza il nostro focus è quello di massimizzare il fan-out.

Dato che la index entry è fatta come <search_key_value, page_pointer>, la sua grandezza dipende quasi intersamente dal valore della search key. Grazie alla

key compression possiamo ridurre la lunghezza delle stringhe nella search key per continuare a ridirezionare bene nei nodi ma tenere anche un fan-out alto.

Bulk-Loading Fare bulk-loading su un B+Tree vuol dire creare un albero da dati pre-esistenti. Di seguito il processo:

1. Sort delle data entry sulla search key
2. Allochiamo una pagina vuota per la root e inseriamo un puntatore alla prima pagina delle entry sorted
3. Aggiungiamo una entry alla root page per ogni pagina delle data entry sorted, e procediamo fino alla fine in questo modo

Costo totale: (R+E) I/Os.

Capitolo 11: Hash-Based Indexing

Utilizzare le funzioni di hash per fare index: molto buone per le equality search ma non supportano affatto le range search e per questo non vengono solitamente utilizzate nei db commerciali.

Static Hashing

- Le pagine che contengono le data entry sono collezioni di **buckets** con una pagina primaria ed una lista di overflow. Un file dunque è composto da N buckets con una pagina primaria per file.
- Per cercare una data entry applichiamo una funzione di hash h per identificare il bucket corretto e poi cercarla nel bucket. (1 I/O)
- Per inserire una data entry applichiamo la funzione di hash h per identificare il bucket dove va l'elemento. Se è pieno, lo agganciamo alla lista di overflow. (2 I/O)
- Per rimuovere una data entry facciamo la stessa cosa. Se la sua lista di overflow è vuota, la pagina rimanente è tolta dalla lista di overflow e messa nella lista delle pagine free. (2 I/O)
- Il numero dei bucket è fissato, e questo determina quasi gli stessi problemi di ISAM, ma anche gli stessi vantaggi.

Extendible Hashing (Dynamic Hashing)

- Utilizziamo una directory di puntatori ai buckets e raddoppiamo la grandezza del numero dei buckets semplicemente raddoppiando la directory e smezzando il bucket che sta facendo overflow
- Raddoppiando la directory, invece che semplicemente il buckets, consumiamo meno spazio per fare la stessa identica cosa che raddoppiare l'intero file.
- La tecnica base utilizzata è quella di trattare il risultato dell'applicazione della funzione h come un numero binario ed interpretare gli ultimi d bit, dove d dipende dalla grandezza della directory stessa, perchè è un offset

della stessa. d è detta la **profondità globale** (global depth) del file hashed, ed è mantenuta nell'header del file, come metadato. Essa viene utilizzata tutte le volte che dobbiamo trovare qualche data entry.
- Non tutti i bucket split necessitano di un raddopio della directory, ma se il bucket smezzato diventa pieno allora dobbiamo raddoppiare la directory di nuovo
- Quando facciamo il raddoppio della directory? Basta tenere conto della **profondità locale** (local depth) per ogni bucket. Se esiste un bucket con la local depth == global depth e stiamo splittando quest'ultimo, allora dobbiamo certamente raddoppiare la grandezza della directory.

Inizialmente tutte le depth locali sono uguali alla global depth, per questo ogni split risulta all'inizio in un raddoppiamento della directory, risultando in un aumento della singola local depth. Intuitivamente se un bucket ha la local depth l, allora i valori hashati di quel bucket finiranno tutti con gli ultimi l bits.

Quando cancelliamo un elemento spesso viene cancellato e basta, anche se potremmo unire i bucket rimanenti, diminuendo la local depth dei bucket.

Problema: la directory cresce eccessivamente e può diventare problematica per le distribuzioni di dati *skewed*! Dato che ci porterebbe a frequenti collisioni nell'hashing, portandoci di nuovo all'utilizzo di liste di overflow quando necessario.

Linear Hashing (Dynamic Hashing)

- Non abbiamo bisogno delle directory, al contrario dell'extendible hashing.
- Può naturalmente evitare le collsioni, si aggiusta da solo per quanto riguarda inserzioni e cancellazioni e i suoi split times sono molto flessibili.
- Se la distribuzione è skewed, il linear hashing è peggiore dell'extendible hashing
- Utilizziamo una *famiglia di funzioni di hashing* h0...hn con la proprietà che il range di ogni funzione è doppio del successivo: se h_i mappa una data entry in M bucket, $h_(i+1)$ mappa una data entry in 2M bucket (molto spesso viene utilizzata l'aritmetica modulare per la scelta di questa funzione)
- Lo split viene fatto in *round*. Durante lo splitting round i, solo le funzioni di hashing h_i e la sua successiva vengono utilizzate, e così via.

Un contatore *level* viene utilizzato per dire quale è il round corrente, ed è 0 di default. Il bucket da splittare è denotato come *next* ed è inizialmente 0, ovvero il primo. Denotiamo il numero di bucket nel file all'inizio del round Level L come N_L. Quando triggheriamo uno split il bucket *next* è splittato e la funzione di hash $h_(L+1)$ ridistribuisce i valori tra il bucket e la sua immagine di split. Dopo aver splittato un bucket, *next* viene incrementato, e così via. Se il bucket puntato da *next* è pieno, e dobbiamo fare un'inserzione, allora facciamo uno split senza aver bisogno di un bucket di overflow.

Extendible vs Linear hashing

- Per distribuzioni uniformi: Linear Hashing > Extendible Hashing
- Per distribuzioni skewed: Extendible Hashing > Linear Hashing

Capitolo 13: External Sorting

Quando è che il DBMS fa sorting?

1. Quando l'utente fa una query che richiede una risposta sorted
2. Come primo passo del bulk loading di un tree index
3. Quando eliminiamo copie duplicate in una collezione di record
4. Quando facciamo join

Two-way Merge Sort

- Questo algoritmo utilizza unicamente **3 pagine** nella memoria principale ed è troppo semplice ed inefficace per essere utilizzato nei DBMS, è stato visto solo a scopo educativo
- Ogni sub-file sorted è detto una **run**
- Anche se l'intero file non ci sta nella main memory, possiamo *spezzettare* il file in sub-file, fare il sorting su quelli per poi fare un merge di quelli quando vogliamo in un secondo momento. Per ogni sub-file, possiamo utilizzare algoritmi in-memory come quicksort per poi fare la write a fine pass, risultando in un costo di 2 I/O per pagine, per pass.
- Questo algoritmo, anche se utilizzassimo più di 3 pagine in main memory, non le utilizzerebbe efficacemente, e per questo nasce l'external merge sort

External Merge Sort

Con B pagine di buffer disponibili e N pagine di un file, vogliamo fare un sort. Dobbiamo cercare di fare più pass cercando però di minimizzarne il numero. Nella pass 0, leggiamo B pagine e facciamo un sort interno per produrre N/B run di B pagine ciascuna (tranne per l'ultima run che potrebbe contenere meno pagine) Nelle pass successive, utilizziamo B-1 pagine di buffer per l'input e il resto per l'output, quindi effettivamente facendo B-1 merge per ogni pass, come mostrato nell'immagine.

Costo:

- Numero di passate: $1 + \log_{B-1}(N/B)$
- Totale I/O: (I/O per passata) * (# di passate) = $2N(1 + \log_{B-1}(N/B))$
- CPU-cost maggiore ma I/O cost minore, dunque un netto miglioramento dato che le task in CPU sono nettamente preferibili
- Per minimizzare il numero di run potremmo utilizzare una variante aggressiva di sorting esterno chiamato **replacement sort**, andando a fare un'output medio di 2B pagine sorted.

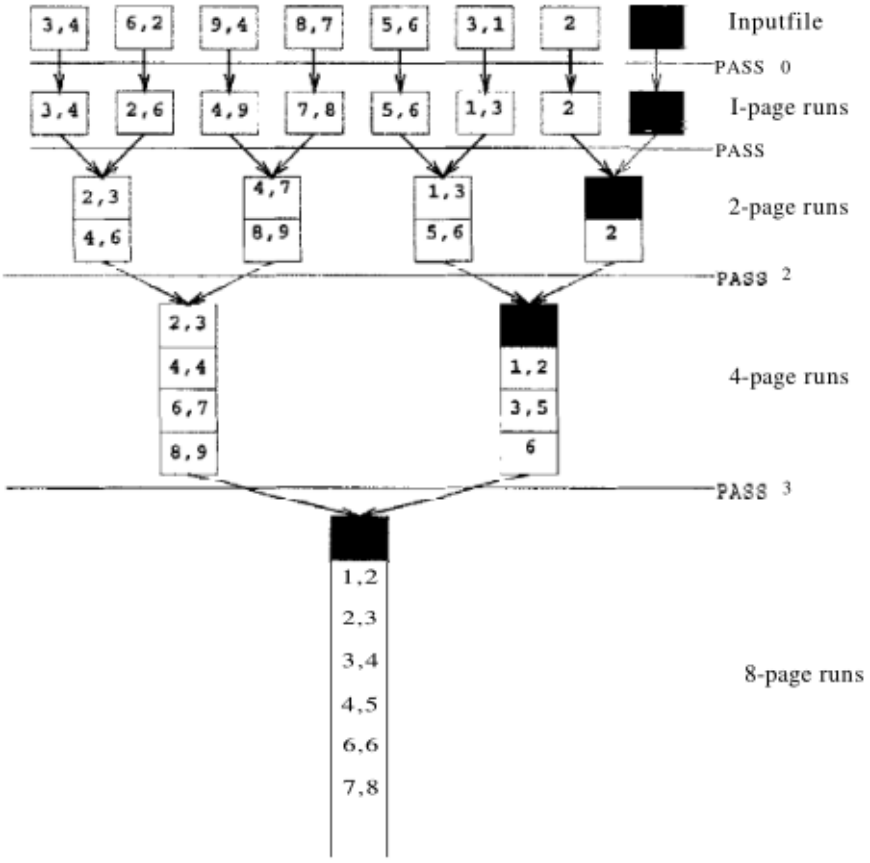

Figure 13.2 Two-Way Merge Sort of a Seven-Page File

Figure 5: Two-way merge sort

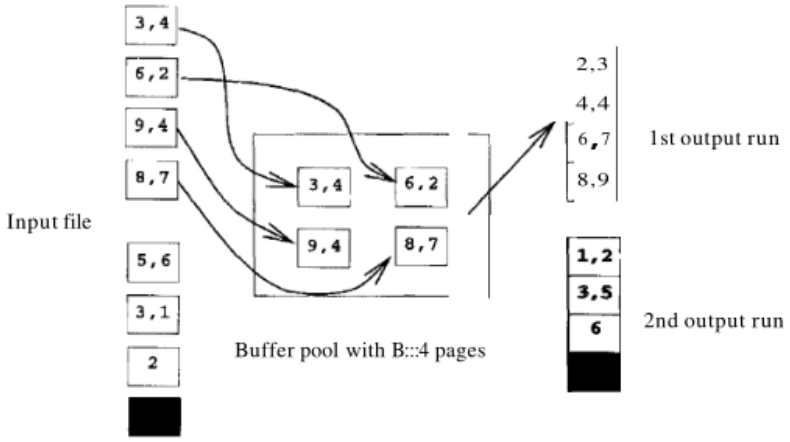

Figure 13.4 External Merge Sort with B Buffer Pages: Pass 0

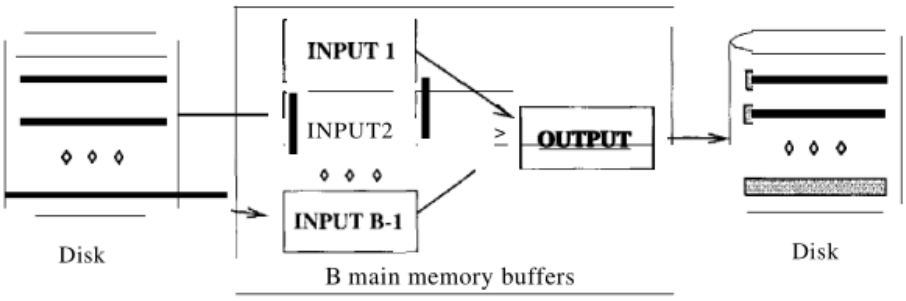

Figure 13.5 External IVlerge Sort with B Buffer Pages: Pass $i > 0$

Figure 6: External merge sort, pass 0 e successivi

Replacement Sort Iniziamo leggendo le pagine del file delle R tuple, fino a quando il buffer diventa pieno, riservando una pagina per l'utilizzo dell'input necessario dopo e una per l'output. In questo momento chiamiamo le B-2 pagine delle R tuple il **current set**. L'idea è quella di ripetitivamente prendere le tuple nel current set con il valore k minore (dove k sarebbe il valore della chiave di ricerca k), che sarebbe comunque più grande del più grande valore k dell'output buffer e fare un'append all'output buffer. Per far sì che l'output buffer rimanga sorted, la tupla scelta tra R deve soddifare la seguente condizione: k >= $k_o b$. Di tutte le tuple che soddisfano questa condizione prendiamo solamente quella con il k minore. Spostando questa tupla nell'output buffer lasciamo uno spazio vuoto nel current set, che sarà riempito con il prossimo input.

Quando terminiamo, però, una run per iniziarne una prossima?

Fino a quando qualche tupla t nel current set ha un k più grande del k della tupla più recentemente messa in append, possiamo semplicemente fare append di t all'output buffer e la run corrente può essere estesa. Quando ogni tupla nel current set è più piccola della più grande tupla nell'output buffer, l'output buffer è scritto e diventa l'ultima pagina nella run corrente. Iniziamo in questo modo la run successiva ed il ciclo continua. Questo algoritmo genera così in media run di grandezza 2B.

Double Buffering e I/O blocking

Il goal è minimizzare il numero di passate nell'algoritmo di sorting dato che ogni pagina nel file da leggere è letta e scritta ad ogni passata, dunque ha senso *massimizzare il fan-in* durante il merge allocando solo una buffer pool per run e una pagina di buffer per il merge dell'output. In questo modo possiamo fare merge di B-1 run, dove B è il numero di pagine nella buffer pool. Dunque è meglio in questo caso leggere e scrivere *in blocchi* piuttosto che in pagine.

Leggiamo e scriviamo dunque in *buffer blocks* di b pagine. Se scegliamo di fare merge di buffer blocks grandi, il numero di passate aumenta, aumentando perciò il numero di I/O di pagina.

Internal Sort: Heapsort

Utilizziamo qui una coda di priorità (heap) in cui si tengono i dati ordinati in modo tale che il costo del sort è ammortito dalla costante presenza della struttura.

- Nel caso migliore si crea un ramo lungo N e la situazione è già ordinata
- Caso medio: si creano run lunghi 2B
- Caso peggiore: si fa tutto il calcolo per ottenere run di lunghezza B (e qui è più efficiente utilizzare il quicksort)

Capitolo 14: Query Evaluation

Le tecniche usate per valutare il costo degli operatori relazionali sono le seguenti:

- Indicizzazione
- Iterazione
- Partizionamento
- Utilizzo di statistiche e cataloghi

Selezione semplice

Per fare una semplice selection siamo portati per la sua struttura ad utilizzare una delle tre:

1. Sequential Scanning (2 frame di buffer per input ed output)
2. Index Scan: se la condizione è uguaglianza scegliamo SH, se è range facciamo una scansione sequenzale e
3. Se l'indice è clustered facciamo SH, si arriva all'inizio del range di intervallo e si va sequenzialmente dopo fino alla fine
4. Altrimenti facciamo SH sull'indice e poi facciamo SS sulle foglie dell'indice dato che le chiavi ordinate sono tutte lì
5. Hashing, utilizzato solo in caso di query per uguaglianza

Selezione complessa

Ovvero una selezione che include una congiunzione o disgiunzione con altre condizioni. I fattori che entrano in gioco nei costi qui sono:

1. Accesso sequenziale o meno
2. Indice clustered o unclustered
3. Selettività del predicato, che si stima grazie al **catalogo**
4. Utilizzo di un indice

Proiezione

Implementazione: SS con 2 frame di buffer

Si applicano gli stessi riguardi e calcolo di costi dei precedenti

Join

L'operazione più costosa tra tutte. Ipotizziamo di lavorare con due relazioni R ed S, le cui cardinalità sono importantissime. Valuteremo ora i seguenti modi per fare join passo per passo:

1. Simple Nested Loop Join
2. Index Nested Loop Join
3. Block Nested Loop Join
4. Sort-Merge Join
5. Sort-Merge Join Refined

6. Hash Join

Nota sui costi:

- p_R: numero di record per pagina
- M: numero di pagine in R
- N: numero di pagine in S

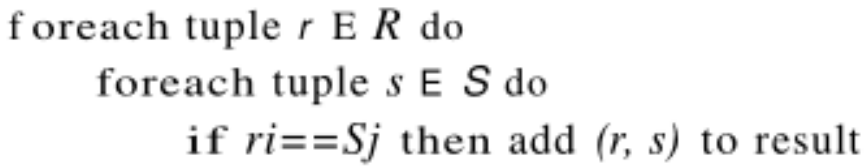

Figure 7: Simple Nested Loop Join

Simple Nested Loop Join

- Non si tratta di un algoritmo blocking
- Costo: $M + p_R * M * N$ (tuple-oriented: meglio minimizzare le tuple della relazione interna)
- Costo: $M + MN$ (page-oriented: meglio minimizzare le tuple della relazione esterna)

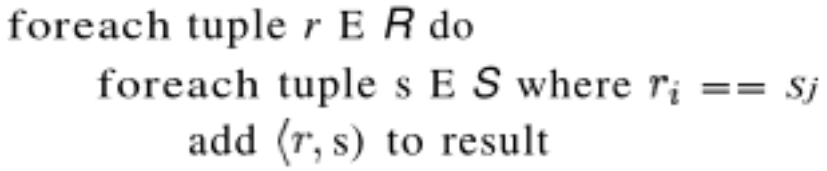

Figure 8: Index Nested Loop Join

Index Nested Loop Join

- Ciascun record della relazione esterna comporta 1 accesso all'indice, quindi il costo di questa operazione è fortemente condizionato dal numero di volte a cui ci accediamo.
- Se l'indice è *clustered* allora facciamo 1 I/O ad accesso, mentre per *unclustered* parliamo di 1 I/O per tupla di S che fa match
- Costo: $M + ((M + p_R) * X)$ dove X è il costo per trovare le tuple di S che matchano

Block Nested Loop Join

- Lavoriamo a livello di blocchi di pagine: dato un numero B di pagine di buffer (+ 1 per output), possiamo tenere un numero minore od uguale a

```
foreach block of B-2 pages of R do
    foreach page of S do {
        for all matching in-memory tuples r ∈ R-block and s ∈ S-page,
        add (r, s) to result
    }
```

Figure 9: Block Nested Loop Join

B-2 per la relazione esterna. Le due pagine rimanenti vanno allocate per la relazione interna e per l'output.

- Portiamo in memoria blocchi di B-2 pagine e questo ci porta a ciclare sulla relazione interna molte meno volte, ovvero un numero di volte pari al numero di blocchi che siamo riusciti a formare (ad es. se abbiamo un file da 80 pagine e ne riusciamo a portare 10 in memoria cicliamo $80/10 = 8$ volte).

Possono capitare diverse situazioni:

1. Carichiamo in memoria B-1 pagine di R e 1 di S, con 1 frame riservato all'output. Il fattore è pari a B-2, dato che la nostra pagina interna considerata si riesce a confrontare con tutte quelle del blocco considerato
2. Carichiamo B-1 pagine di S ed 1 di R, 1 frame per output. Riusciamo qui ad utilizzare la politica MRU su tanti cicli, ma lavoriamo di più nella memoria interna

Quello che facciamo tipicamente: con più frame di buffer cerchiamo un compromesso (tipo metà R e metà S), così facendo riduciamo il numero di cicli su S e i page fault, utilizzando politica MRU se si tratta di letture sequenziali.

Sort Merge Join Nota: *Funziona solo con i join per uguaglianza*

- Si tratta di un algoritmo bloccante, non si può fare il match di una relazione prima che la si *conosca* completamente: devo aspettare l'operazione di ordinamento per procedere, ovvero abbiamo una grande debolezza
- Sfruttiamo il fatto che le relazioni siano ordinate prima di fare il join
- Nella pratica il costo della fase di merge è 1 scan per relazione

Costo totale: Sorting + Merging (come costo ci ritroviamo sullo stesso path del Block Nested Loop Join), ovvero, $O(Mlog(M) + Nlog(N))$

Sort Merge Join Refined Combiniamo simultaneamente la fase di ordinamento con quella di merge.

- Possiamo creare delle sotto-run ordinate della grandezza di 2B grazie al Replacement Sort
- Abbiamo ora $\sqrt{L/2}$ run per ogni relazione, così facendo abbiamo meno run di B e possiamo combinare le fasi di merge senza buffer addizionali

- Questo ci consente di effettuare il sort merge join al costo di leggere e scrivere R e S nella prima passata e R e S nella seconda passata.

Costo totale: $3(M + N)$

Hash Join Nota: *Funziona solo con i join per uguaglianza*

Come per il Sort-Merge Join utilizziamo partitioning per identificare le partizioni nella fase di partitioning e nella fase di probing andiamo a testare le condizioni di equality join. Al contrario di Sort-Merge non facciamo sorting ma hashing.

- L'idea è quella di fare hashing di entrambe le relazioni sull'attributo di join, utilizzando la stessa funzione h
- In pratica costruiamo un hash table in-memory per la partizione di R, utilizzando un *h2* tale che $h2 \neq h$, per ridurre costi CPU
- Abbiamo bisogno abbastanza memoria per mantenere la tabella di hash, che è più grande della partizione di R

Costo totale: $3(M + N)$ se assumiamo che le partizioni stiano in memoria nella seconda fase. Se i dati sono molto skewed, allora è meglio sort-merge refined.

Capitolo 15: Query Optimization

Consultare il capitolo sul libro.

Capitolo 16: Transaction Manager

- Transazione: una serie di azioni di tipo read o write di oggetti database. Una transazione può finire con una commit oppure con un abort.
- Schedule: una lista di azioni da un set di transazioni. Uno schedule è *complete* quando contiene un abort oppure una commit per tutte le transazioni listate. Uno schedule è *seriale* quando le azioni di transazioni differenti non sono interfogliate. Vogliamo un'esecuzione concorrente dato che molto spesso la CPU aspetta le operazioni I/O.
- L'effetto di una schedule serializzabile è lo stesso che eseguire in momenti diversi le due transazioni in oggetto

Conflitti

- Write-Read: T2 legge qualcosa scritto da T1 (dirty read)
- Read-Write: il contrario (unrepeatable read)
- Write-Write: T2 scrive su qualosa scritto da T1 (blind write –> Lost Update)

Uno **schedule serializzabile** su un set S dunque è uno schedule i cui effetti su un database consistente sono garantiti essere gli stessi di qualche schedule complete seriale sul set di transazioni committed di un certo set S.

Uno **schedule recuperabile** è uno schedule dove le transazioni fanno commit se e solo se tutte le transazioni che hanno dovuto leggere fanno commit. Se una transazione legge solo i cambiamenti di transazioni committed, non solo abbiamo una schedule recuperabile ma evitiamo anche gli **aborti in cascata**.

Strict 2PL

Lo strict 2PL è un protocollo di locking che ha due regole principali: 1. Se una transazione T vuole leggere (o modificare) un oggetto, deve prima prendere un lock shared (rispettivamente, exclusive) sull'oggetto 2. Tutti i lock tenuti da una transazione vengono rilasciati quando la transazione viene completata

Ovviamente, protocolli di locking (come questo) possono portare a situazioni di deadlock

Nota: lo strict 2PL ammette solo schedule *conflict serializable* (tra pochissimo la spiegazione) perchè:

- Una schedule *conflict serializable* ha il grafo aciclico
- Strict 2PL ammette solo schedule con grafi aciclici

`Maggiori dettagli su 2PL ed esempi si possono trovare sul libro`

Performance di locking

Gli schemi di locking utilizzano due strumenti: blocking e aborting. Entrambi i meccanismi ovviamente diminuiscono la performance del nostro sistema. Le transazioni bloccate potrebbero trattenere il lock che forza le altre transazioni ad aspettare, mentre le transazioni abortite devono ripartire da capo, impegnando una grande porzione di tempo.

Trashing: fenomeno di riduzione di throughput dovuto ad un aumento della concorreza delle transazioni sul sistema. Questo fenomeno si riduce andando a limitare la concorrenza delle transazioni ad un certo livello prefissato.

Come aumentare il throughput?

1. Bloccando gli oggetti molto piccoli (riducendo perciò la probabilità che a due transazioni servano effettivamente quell'oggetto)
2. Riducendo il tempo che una transazione può tenere un certo lock
3. Riducendo gli **hot spot**, oggetti database che sono tenuti frequentemente e modificati con egual frequenza.

Capitolo 17: Concurrency Control

2PL, Serializzabilità e recuperabilità

- Due schedule si dicono **conflict equivalent** se utilizzano lo stesso set di azioni delle stesse transazioni e il loro ordine di ogni paia di azioni che conflittano è lo stesso. Se due schedule sono conflict equivalent è

facile notare che avranno lo stesso effetto sul database, e quindi possiamo semplicemente fare swapping delle azioni non-conflittuali delle due schedule in modo che l'outcome non venga alterato.
- Una schedule è **conflict serializable** se è *conflict equivalent* a qualche *schedule seriale*. Ogni schedule *conflict serializable* è serializzabile e se il database non si espande o diminuisce di dimensione nessun item viene aggiunto o tolto. In ogni caso non tutte le schedule serializzabili non sono *conflict serializable*.

2PL

Rilassiamo la seconda regola di strict 2PL nel modo seguente: *una transazione non può richiedere lock aggiuntivi una volta che rilascia QUALSIASI lock*. Dunque possiamo osservare come ogni schedule che rispetta 2PL ha una fase di *crescita* ed una fase di *decrescita*.

2PL assicura, come strict 2PL, grafi aciclici.

Una schedule è detta **strict** quando un valore scritto dalla transazione T non è letto o sovrascritto dalle altre transazioni fino a quando T non fa abort o commit.

View Serializability

Due schedule S1 e S2 sono dette view equivalent sulle loro transazioni sse:

1. Se Ti legge il valore iniziale dell'oggetto A in S1, allora deve anche leggere il valore iniziale di A in S2
2. Se Ti legge il valore di A scritto da Tj in S1, allora deve anche leggere il valore di A scritto da Tj in S2
3. $\forall A$ la transazione che fa l'ultima write su A in S1 deve anche fare l'ultima write di A su S2

Una schedule è detta **view serializable** quando è *view equivalent* a qualche schedule seriale. Ogni schedule *conflict serializable* è *view serializable* ma l'inverso non è vero!

Lock Management:

- **lock manager**: parte del DBMS che mantiene traccia dei lock. Possiede una **lock table** che consiste in una hash table con un *data object identifier* come chiave. mantiene anche una **transaction table** (fa esattamente quello che dice di essere).
- Una lock table entry può essere una pagina, un record... contiene le seguenti informazioni: il numero di transazioni che hanno un lock, che lock è ed un puntatore alla lista delle richieste del lock.

Quando un *lock manager* riceve una richiesta di lock vengono fatte le 3 seguenti cose:

1. Se è richiesto uno *sl*, la coda delle richieste è vuota e l'oggetto non è tenuto da un *xl* allora il lock manager dà il permesso ed aggiorna la lock entry con quella richiesta
2. Se è richiesto un *xl* e sono soddisfatte le cose di cui sopra, allora il lock manager rilascia il permesso e aggiorna la lock entry con quella richiesta
3. In ogni altro caso il lock non può essere dato e la richiesta è aggiunta alla lista delle richieste per quello specifico oggetto, la transazione è *sospesa*

Nota: lock ed unlock sono implementate come operazioni atomiche, tipo con un semaforo. In addizione ai lock, spesso vengono anche usati *latch* e *convoys*.

Un lock può essere upgradato oppure downgradato (sl to xl oppure xl to sl)

Deadlock prevention

Un modo per assegnare la priorità ad una transazione è assegnare ad essa un **timestamp**. Quando una transazione Ti richiede un lock ma Tj ha un lock conflittevole, il lock manager può scegliere tra una delle due policy:

1. **Wait-Die**: se Ti ha priorità più alta, allora può aspettare. Altrimenti si fa abort.
2. **Wound-Wait**: se Ti ha priorità più alta, Tj fa abort. Altrimenti Ti aspetta.

Dobbiamo anche far si che nessuna transazione sia perennemente in abort dato che non ha abbastanza priorità.

Conservative 2PL

Una variante di 2PL che riesce anche a prevenire i deadlock. Una transazione ottiene tutti i lock che le serviranno prima di effettuare qualsiasi operazione. Ovviamente bloccare tutti i lock in questo modo fa aumentare abbastanza i tempi e diminuisce di molto la concorrenza ma evita del tutto i deadlock nel sistema.

Index & Predicate Locking

Possiamo fare locking sull'index page per evitare che avvengano particolari conflitti (ad es. blocchiamo la index page rating=1). L'index locking è solamente un caso speciale di **predicate locking**, che però sarebbe troppo costoso da implementare e quindi quasi mai utilizzato.

Locking nei B+Tree

Come possiamo bloccare in modo efficiente una particolare foglia? Utilizziamo la tecnica del **lock-coupling** (o crabbing), che consiste nel procedere dalla radice alla foglia facendo locking sul figlio e rilasciando il parent mentre andiamo avanti.

Multiple-granularity lock

Strategia di locking che ci permette di mettere lock su oggetti che contengono altri oggetti, non è da confondere con i B+Tree locking, nonostante ci si faccia intimidire dalla definizione che potrebbe rientrare nella categoria.

Possiamo vedere il DB e le sue singole transazioni come un albero e fare locking sulle singole relazioni (alta concorrenza, molti lock) oppure sulle parti più alte dell'albero, come le transazioni T (meno lock, ma meno concurrency)

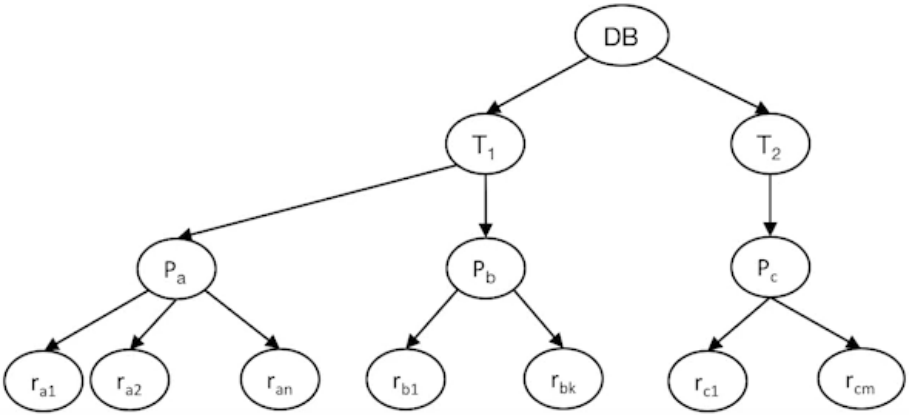

Figure 10: Multiple-granularity lock

Il MGL utilizza:

- Intention Shared locks
- Intention Exclusive locks

Entrambi conflittano con i lock esclusivi mentre gli intention exclusive anche con gli exclusive classici.

Quando decidiamo di fare lock sulla parte alta della gerarchia, dobbiamo mettere un IS-IX lock sulle parti più basse (i figli)

MGL deve essere utilizzato insieme a 2PL per assicurarsi la serializzabilità. 2PL dice quando i lock devono essere rilasciati. In quel momento, i lock ottenuti utilizzati con MGL deveono essere rilasciati dalla foglia alla radice. 2PL viene utilizzato con una matrice di compatibilità nel momento del rilascio.

Optimistic Concurrency Control

Tipo di protocollo che non ha bisogno di locking e che tiene un approccio ottimistico sulle transazioni. Di base, per il protocollo ottimistico, la maggior parte delle transazioni non fa conflitto con le altre transazioni. Le transazioni procedono in 3 fasi dunque:

1. Read: la transazione fa l'esecuzione, leggendo i valori dal database e scrivendoli su un workspace privato
2. Validation: nel suo workspace privato viene validata la transazione, facendo controllo che non abbia potuto conflittare con qualsiasi altra transazione.
3. Write: se la validation determina che non ci possano essere conflitti, il private workspace viene copiato nel db.

Ad ogni transazione viene assegnato un timestamp TS(Ti) all'inizio della fase di validazione, e nella fase viene controllato anche la serializzabilità temporale della stessa. Per ogni coppia di transazioni Ti e Tj con timestamp TS(Ti) e TS(Tj) una delle tre deve verificarsi per passare:

1. Ti completa tutte le fasi prima che Tj inizi
2. Ti completa prima che Tj inizi la sua fase Write e Ti non scrive niente che Tj legga
3. Ti completa la sua Read prima che Tj completi la sua Read e Ti non scrive nessun oggetto che è letto o scritto da Tj

Ognuna di queste condizioni assicura che non si verifichi un conflitto tra le due. La dimostrazione è disponibile a pagina 567.

Improved Conflict Resolution

OCC utilizza 3 regole che bisogna verificare per ogni paia di transazione, rendendo così lenta la verifica per ogni transazione. Inoltre, crea un overhead non necessario con la quantità di volte che fa ripartire o fa abortire una transazione. Per ultima cosa, presenta la possibilità di falsi positivi, che vediamo ora.

Non abbiamo nessun modo per dire quando Ti ha scritto l'oggetto al tempo in cui facciamo la validazione di Tj, visto che tutto quello che abbiamo è una lista di oggetti scritti da Ti e letti da Tj. Questi falsi-conflitti che nascono dalla situazioni di cui sopra possono essere alleviati da meccanismi più delicati e simili al locking. Procediamo.

Prima di leggere un dato, la transazione T scrive un **access entry** in una tabella di hash. La *access entry* è del tipo *<id, object_id, modified_flag=false>* e le entry sono hashate sull'object_id. Otteniamo poi un XL temporaneo sull'hash bucket che contiene la entry e il lock è mantenuto durante la copia del dato dal database buffer al workspace primato della transazione.

Durante la validation di T gli hash bucket a cui facciamo accesso per T sono lockati con XL per controllare che T non faccia conflitto. T fa conflitto se la flag *modified* è **true** in una delle access entry.

Se T è validato, facciamo una festa di compleanno a T, lockiamo tutti gli hash bucket di tutti gli oggetti modificati da T, prendiamo tutte le access entry per l'oggetto, mettiamo la *modified* a **true** e rilasciamo il lock sul bucket. Se usiamo la policy *kill*, le transazioni che sono entrate nelle access entry vengono riavviate.

Nella maggior parte dei casi usare la policy *kill* è più veloce della policy *die*, perchè riduce i tempi di risposta ed attesa. Ma con la *kill* facciamo meno I/O generale per le read.

Timestamp-Based Concurrency Control

Utilizziamo il timestamp come segue: ad ogni transazione assegnamo un timestamp all'inizio. Ci preoccupiamo solo poi, al tempo di eseguirla, se l'azione Ai di Ti conflitta con l'azione aj di Tj, dato che ai< aj ovviamente. Facciamo abort e restartiamo.

Per fare ciò ci servono un RTS (read timestamp) e WTS. Se una transazione T vuole leggere O: TS(T) < WTS(O) violerebbe l'accesso e quindi abortiamo la transazione e la riavviamo.

Quando vogliamo scrivere, invece:

1. Se TS(T) < RTS(O) allora abort and restart
2. Se TS(T) < WTS(O), vorremmo fare abort e restart ma invece applichiamo la **Thomas Write Rule**
3. In ogni altro caso, T scrive O e WTS(O) diventa TS(T).

Thomas Write Rule

Se TS(T) < WTS(O) allora la write sull'oggetto è più recente di quella proposta e quindi la possiamo tranquillamente ignorare. Possiamo interpretarla come una write avvenuta immediatamente prima di questa e non è mai stata mai letta da nessuno. Tutto in regola.

Migliorare il TBCC con la Recoverability

Il Timestamp-Based Concurrency Control appena visto permette delle schedule non-recuperabili. Possiamo modificarlo per non permetere queste semplicemente facendo **buffering** di tutte le write fino a quando la transazione non fa commit. Questo viene spesso usato in sistemi distributi invece che in sistemi centralizzati.

Multiversion Concurrency Control

Protocollo similare che arriva alla serializzabilità. Il goal è quello che una transazione non deve aspettare a leggere un oggetto del db e l'idea è quella di mantenere differenti versioni dell'oggetto in tale db.

Se una transazione Ti vuole scrivere un oggetto, dobbiamo assicurarci che l'oggetto non sia stato già letto da qualche altra transazione Tj tale che TS(Ti) < TS(Tj). Per fare check di questa condizione, ogni oggetto ha anche un read timestamp associato e quando la transazione fa una read, aggiorniamo l'appena citato con l'ultima read. Se Ti vuole leggere O e TS(Ti) < RTS(O), abort e restart. Altrimenti, Ti crea una **nuova** versione di O e mette la read e la write timestamp a TS(Ti) appena avvenuto.

Capitolo 18: crash recovery

Introduciamo ARIES, un algoritmo di recovery. ARIES utilizza una politica steal, no-force. I restart procedono in 3 fasi:

1. **Analisi**: identifica le pagine sporche nel buffer pool e le transazioni attive a tempo di crash
2. **Redo**: ripete tutte le azioni partendo da un punto appropiato del log e ripristina il database allo stato del crash
3. **Undo**: fa undo delle transazioni che non hanno fatto commit

ARIES utilizza 3 principi:

- **Write-Ahead Logging**: ogni cambiamento nel database viene loggato ed il log viene scritto nel db PRIMA che l'azione venga scritta
- **Repeating-History** durante il *Redo*: al restart, tutte le azioni vengono ripetute prima del crash, anche quelle committate, e dopo fa undo delle transazioni che erano attive a tempo di crash.
- **Logging dei cambiamenti durante l'undo**: serve più che altro se ci sono crash seriali

Log

Ogni azione del log ha il suo **LSN**, o log sequence number, assegnato monotonicamente in ordine. Altri dati LSN che mantiene:

- pageLSN: il lsn del log record più recente del cambiamento di una pagina
- prevLSN: link al log precedente, dato che stiamo lavorando con una linked list come log
- transID: id della transazione T
- type: tipo della transazione (update, commit, abort, end, undo)
- flushedLSN: neanche goku lo sa a cosa serva questo coso

Un log record viene scritto in ognuno di questi casi:

1. Page update
2. Commit
3. Abort
4. End
5. Undo di un update (si scrive un **Compensation Log Record** che contiene un *undoNextLSN*, settato al *prevLSN* dell'azione su cui si fa undo)

Altre strutture per il recovery

- **Tabella delle transazioni**: contiene una entry per ogni transazione attiva, commited o aborted, ed il suo lastLSN.
- **Dirty page table**: abbastanza autoesplicativo. Contiene un field detto il **recLSN** che è il LSN del primo log record che ha portato la pagina ad essere sporcata.

Checkpointing

Un checkpoint è uno snapshot di sistema. I checkpoint vengono tenuti nel log con i record di tipo *begin_checkpoint* oppure *end_checkpoint*. Fare checkpointing aiuta subito dopo un restart perchè per selezionare da dove ripartire, ripartiremo sicuramente dall'ultimo checkpoint.

ARIES: Analisi

Durante la fase di analisi vengono effettuate le seguenti operazioni:

1. Viene determinato da dove ripartire con il Redo
2. Viene determinato un set di pagine nella buffer pool che erano dirty a tempo di crash
3. Vengono identificate le transazioni che erano attive a tempo di crash e devono essere disfatte.

Nello specifico l'iterazione è la seguente:

1. Esaminiamo il *begin_checkpoint* più recente, inizializzando le due strutture di recovery di cui abbiamo appena parlato.
2. Facciamo la scansione del log dall'inizio del punto settato fino alla tail nel modo seguente:
    1. Se incontriamo un record *end_log*, T è rimosso dalla tabella delle transazioni.
    2. Se incontriamo un'altro log record, aggiungiamo una entry alla tabella delle transazioni. Se T inoltre è modificata:
        - il lastLSN è impostato al LSN del record
        - se il log record è un commit, mettiamo lo status a C, altrimenti a U.
    3. Se il log record è *redoable*, e P analizzato non è nella tabella delle pagine sporche, aggiungiamo una entry con page_id P e recLSN = LSN del record *redoable*
3. Alla fine dell'analisi, la tabella delle transazioni contiene una lista di tutte le transazioni attive al tempo del crash

ARIES: Redo

Durante questa fase ARIES riapplica gli update di *tutte* le transazioni, che siano committed oppure aborted. Si veda Repeating-History per maggiori dettagli.

Iniziamo con il log record che ha il più piccolo *recLSN* di tutte le pagine nella tabella delle pagine sporche. Partendo da qui, scansioniamo fino alla fine del log. Per ogni record *redoable*, Redo controlla che sia veramente rifacibile e lo fa a meno che una di queste condizioni non si verifichi:

1. La pagina non è nella tabella dirty (ovvero, tutti i cambiamenti sono già nel disco)

2. La pagina è nella dirty ma recLSN > LSN corrente (ovvero, l'update è propagato nel disco)
3. pageLSN >= LSN corrente (ovvero, l'update è propagato nel disco a colpa di un update corrente o precedente)

In ogni altro caso rifacciamo l'azione corrente e pageLSN = LSN corrente.

ARIES: Undo

Guardiamo le **loser transaction** (transazioni attive a tempo di crash) e le disfiamo, sempre guardando il *lastLSN* delle stesse. Intanto, teniamo un set (chiamato ToUndo) che contiene tutti i *lastLSN* delle *loser transaction*.

1. Se undoNextLSN non è null, undoNextLSN è aggiunto a ToUndo, altrimenti, scriviamo un *end* record.
2. Se è un *update* record, scriviamo un CLR e l'azione corrispondente è disfatta, mentre il *prevLSN* è aggiunto a ToUndo
3. Ci fermiamo quando `len(ToUndo) == 0`