Domande/Risposte
Sapino

Premessa:

Elenco di tutte (o quasi) le domande degli ultimi anni (fino a luglio 2022) fatte dalla Sapino nei suoi bellissimi scritti.

Utilissimo per ripassare ed esercitarsi per un esame che è diverso dagli altri, unico, raro…un'esperienza di vita insomma.

Le risposte sono frutto di un lavoro di gruppo, ma non garantisco la totale correttezza.

Riferimento:

Telegram: @Aleeee96

Introduzione DBMS

1. Negli incontri introduttivi abbiamo discusso il fatto che il modello relazionale e i DBMS relazionali possono risultare troppo "general- purpose" per certe applicazioni. Discutete alcuni degli svantaggi del modello relazionale, e le ragioni che vi indurrebbero a optare per soluzioni meno generali, in certe situazioni.

- Tipicamente nei db relazionali vale il concetto di mondo chiuso (è vero tutto quello che è stato esplicitato nel db, se non è scritto nel db non è vero), Una delle ragioni che farebbero optare per altre soluzioni può essere dovuta al fatto che gli interessi di ciascun team di sviluppo che utilizza un dato database, potrebbero entrare in conflitto tra loro e questo può portare a rendere lo schema molto complesso. Il modello relazione quindi si basa su uno schema rigido che deve tener conto di tutte le applicazioni che ci interagiscono, deve essere compatibile per tutte le applicazioni ma potenzialmente non è fatto ad hoc per nessuna di queste. L'approccio 'general-purpose' riduce le performance e complica la realizzazione/manutenzione dello schema. Un modello più specifico e magari schemaless permette di avere una base di dati più flessibile e più 'customizzabile' sulle applicazioni che ci lavorano direttamente, aumentano le performance e riduce i compromessi che le app devono tener conto per interagire sulla base di dati. Un altro problema è l'impedence mismatch in quanto potrebbe essere più comodo avere un database che accetta le strutture dati che il programma contiene; cioè, strutture che non sono solo relazionali, quindi XML o JSON, oppure usando un grafo.

2. Abbiamo iniziato il corso proponendo una carrellata storica sull'evoluzione dei modelli per la gestione dei dati, ribadendo ripetutamente che i DBMS relazionali; tendono ad essere più facilmente ottimizzabili dei DBMS ad oggetti. Alla luce di quanto abbiamo discusso nella restante parte del corso, ed in particolare sull'ottimizzazione e sull'esecuzione delle query, discutere quanto accennato inizialmente. Indicare due ragioni a supporto dell'affermazione iniziale dandone spiegazione.

   (simile) Perché i dbms relazionali sono più efficienti di quelli ad oggetti per le ottimizzazioni ed esecuzioni di query?

- Nei modelli relazionali, (anche per un discorso di prevedibilità), l'ottimizzazione è più facilmente applicabile in quanto l'organizzazione tabulare della relazione è strettamente vicina a livello fisico all'organizzazione dei record nel disco. Cosa invece non vera nei DMBS ad oggetti, perché data la presenza di puntatori diventa difficile fare stime ed ottimizzare di conseguenza. Infatti, il modello ad oggetti si usa di solito per allocazione continua. Inoltre, essendo l'SQL un linguaggio dichiarativo, con meno operatori dell'algebra relazionale, con un numero limitato di implementazioni, c'è quindi una casistica di ottimizzazioni limitata, perciò quando si domanda all'ottimizzatore l'interrogazione, lui sceglierà l'implementazione più ottimizzata per quella query.
  Inoltre, vale il discorso della prevedibilità dei dati, che sono memorizzati in frammenti tra loro omogenei, e so quali e quante pagine devo leggere di una tabella. Non si fa un uso pesante dei puntatori, che renderebbero imprevedibile la posizione dei dati e gli accessi, finché non seguo il puntatore non so dove il dato è allocato. Nel momento in cui non c'è prevedibilità vengono meno le performance perché è complicato memorizzare in maniera efficiente gli accessi al disco (che sono le operazioni più costose).

3. Abbiamo discusso l'evoluzione dei modelli per la gestione dei DBMS sottolineando che - per quanto importante e vantaggioso in numerosi domini di applicazione - il modello relazionale non è adatto nella gestione di dati relativi a conoscenza semantica, e alla memorizzazione di dati relative all'attività in reti sociali. Discutere brevemente le ragioni alla base di questa inadeguatezza.

- Il modello relazionale non è adatto alla gestione di dati relativi alla conoscenza semantica, poiché è un modello poco flessibile, adatto a domini che necessitano di una maggior garanzia della consistenza dei dati ma meno adatto a nature più flessibili e tolleranti. Le reti sociali (ad esempio, i social networks) rispecchiano una struttura reticolare meglio rappresentabile con un modello a grafo dove ogni nodo (concetto) è relazionato con altri medianti archi (relazioni). Dato un nodo che rappresenta un utente, è facile navigare in maniera procedurale per trovare le altre entità in qualche modo sono connesse con l'utente. In questo caso si guadagna in flessibilità, scalabilità e velocità nella navigazione.
  Anche quindi con il paradigma XML otteniamo benefici, poiché molto flessibile, e adatto a gestire una conoscenza di dominio incrementale. Nel modello XML (e in modo specifico nel paradigma RDF) i fatti che descrivono il mondo sono rappresentati come triple <soggetto, predicato, complemento oggetto>, e vengono inseriti incrementalmente nella base di conoscenza ogni volta che viene scoperto qualcosa di nuovo sul mondo.

4. Perché per alcuni tipi di dati non sono adatti database relazionali?

- Perché questi dati possono essere ad esempio, dati con tante relazioni (ontology-like, o anche solo la lista di amicizie di facebook). Questo perché nei db relazioni i join sono operazioni costose dato dal modo in cui vengono salvati i dati (soprattutto se non uso indici). Usando dbms appositi come graph, le relazioni tra nodi vengono salvate fisicamente e di conseguenza accedere un nodo in relazione con un altro, ha costo costante, anche in usi in cui i dati non hanno una struttura fissa.

nosql e database a grafo sono schema-less, rendendo più semplice l'aggiunta dinamica di campi o relazioni sui singoli nodi/documenti. Al contrario dei database relazioni, dove c'è uno schema e tutti i dati devono seguire quello schema.

5. Si può dire che i db ad oggetti sono db NoSQL?

- Non proprio, OSL non è NoSQL ad esempio. I NoSQL sono piuttosto dei linguaggi indicati per i big data (grosse moli di dati, eterogenei, senza schemi), reti sociali ecc., tipicamente vanno bene per i dati di natura gerarchica. I NoSQL sono sistemi in cui tipicamente ci si accontenta di 2 su 4 requisiti ACID, con gli oggetti ci sono ancora.

6. Alcuni Database sono basati su modelli logici molto vicini al modello concettuale dei dati, mentre altri sono più vicini ad un modello fisico. Discutere vantaggi e svantaggi di questi due tipi di modelli logici.

- Un Data Model è un formalismo che consente di descrivere la struttura dell'organizzazione dei dati e i vincoli che i dati devono soddisfare, i vincoli vengono descritti rispetto a un formalismo. La natura dei dati permette di scegliere il formalismo più adeguato. Uno schema di vincoli più vicino ad un modello fisico (come la struttura tabellare del modello relazionale) facilità la gestione del disco in quanto più vicina alla loro memorizzazione in memoria ma consente meno espressività rispetto a uno schema concettuale di vincoli (application level). Il modello concettuale è più flessibile per quanto riguarda la rappresentazione dei concetti ma la flessibilità perde in termini di prevedibilità e soprattutto rende più difficile scrivere delle query efficienti.

Interdipendenza tra moduli

7. Descrivere almeno due casi in cui le scelte alla base del funzionamento di un modulo (buffer manager, disk manager,ecc) ha effetti sensibili su altri moduli a lui collegati.

   (simile) Abbiamo discusso i diversi moduli che costituiscono Un DBMS (disk manager, buffer manager, transaction manager, query optimizer, ecc.), evidenziando il fatto che ciascuno di questi moduli è dedicato alla gestione di uno specifico aspetto della vita del DBMS, e sottolineando al tempo stesso il fatto che non si può immaginare che tali moduli lavorino in totale autonomia. Illustrare due casi specifici in cui le attività dei moduli ad esse dedicati risentono pesantemente delle scelte alla base di altri moduli.

   (simile) Descrivi l'interdipendenza tra i moduli del DBMS e come questi non possono essere sostituiti e alterati senza avere effetti sugli altri. Hanno un impatto locale o globale?

   (simile) Ipotizziamo la modifica/eliminazione di un modulo X, quali possono essere le conseguenze sugli altri moduli?

- Dipende dal modulo che andiamo a eliminare, essendo molto interdipendenti tra loro la rimozione di alcuni di questi causerebbe dei problemi enormi a cascata. Per esempio, se eliminassimo il modulo dei lock causeremmo una marea di errori a cascata a causa dei problemi di parallelizzazione, se eliminassimo il recovery manager non saremmo più in grado di risolvere i danni in seguito a un crash, se eliminassimo il buffer saremmo costretti a operare unicamente su disco causando dei rallentamenti enormi. In caso di modifiche ai moduli il tutto dipende da che moduli andiamo a modificare e come (magari aggiungi uno dei casi elencati alla domanda sotto)

(simile) Abbiamo visto che ciascun DBMS relazionale è costituito da numerosi moduli, ciascuno con le sue funzionalità, strettamente interdipendenti tra di loro. Si supponga di aver osservato che - data l'analisi delle prestazioni del DBMS - potrebbe essere opportuno modificare il modulo per la gestione del buffer. Dire quali altri moduli del DBMS risentirebbero della modifica del buffer manager spiegandone le ragioni. **(fatta alla simulazione)**

- Un esempio di interconnessione tra i diversi moduli potrebbe essere quella tra il:
    - Un esempio è il caso del Recovery Manager che dipende dal Transaction Manager, poiché senza di esso non si avrebbe a disposizione le tabelle (dirty table e transaction table) i checkpoint e il log file da ripercorrere in caso di crash.
    - Un altro esempio è tra il Disk Space Manager che dipende dal Lock Manager, senza di esso la capacità del DBMS di essere scalabile sarebbe rovinata e si rischierebbero continue inconsistenze sui dati
    - Un altro esempio di collaborazione potrebbe essere quello tra il Buffer Manager e il Query Optimizer, infatti, in base alle pagine che ho a disposizione nel buffer, il query optimizer può adottare un approccio rispetto ad un altro. Viceversa, in base al tipo di query, il buffer gestisce le pagine in maniera diversa.
    - Ancora un altro esempio potrebbe essere quello tra il Buffer Manager e il Recovery Module, in caso di Undo/Redo di alcune operazioni.
    - O ancora un altro potrebbe essere quello tra il Buffer Manager e il Transaction Manager, in quanto quest'ultimo (transaction manager) gestisce l'informazione del PIN Count, che serve al buffer manager.
    - Infine, anche tra il Buffer Manager e il Disk Space Manager, in caso di accesso ai dati di una pagina che dal disco viene portata nel buffer.

8. Quali sarebbero le conseguenze in assenza del modulo del buffer e del Buffer Manager

- Andremmo a svolgere tutte le operazioni su disco, il che rallenterebbe notevolmente il sistema danneggiandone le prestazioni; in assenza di buffer, verrebbe meno l'interdipendenza con con il file manager e vado ad annullare l'interdipendenza presente anche con il query optimizer, visto che il numero di pagine presente nel buffer può influenzare la valutazione delle query. Inoltre, caso di crash, non avremmo a disposizione Transaction Table e dirty page table in quanto queste risiedono in RAM

9. Descrivi l'interdipendenza tra il Concurrency Control e il Recovery Manager

- Grazie all'interdipendenza di questi moduli io posso garantire le proprietà ACID, in particolare il recovery manager assicura Atomicità e Durabilità mentre il Concurrency Control assicura Consistenza e Isolamento. L'interdipendenza più importante avviene tra il transaction manager e il recovery manager, poiché il transaction mantiene il log delle operazioni ed effettua i checkpoint delle transaction table e dirty table che sono utilizzate dal recovery manager.

10. Per quali ragioni per la gestione dei file e del buffer in un DBMS non ci si può affidare direttamente alle funzionalità del sistema operativo, ma sono necessari moduli ad hoc?

- Perché un sistema chiuso ci permette di essere più prevedibili ed evita errori che potrebbero essere causati dal sistema operativo. Se prendiamo ad esempio un generico file system classico, questo non provvede ad assicurare la data consistency, non possiede schema per dati strutturati, non provvede a far fuori dati ridondanti. Per esempio, è differente anche la gestione del log di un DBMS rispetto a quella di un Sistema operativo, ognuno implementa la sua strategia particolare. Ma questo perchè il sistema operativo lavora ad un livello diverso. Potrebbe banalmente operare con tecniche base come lru o mru (per la gestione del buffer) o heap o file ordinati (per la gestione dei file). Questo perché non conosce gli aspetti specifici dei database e dei dbms. I dbms hanno

informazioni aggiuntive e quindi possono permettersi tecniche ad hoc più precise come indici o DBMin.

Dischi e File

11. Facendo riferimento al modello dei costi discusso per l'accesso ai file su disco, indicare (spiegandolo) quello che è il costo stimato per la valutazione di una query di uguaglianza (oppure range) nel caso in cui si possa sfruttare un indice B+-tree unclustered che abbia occupazione media dei nodi pari al 70% (oppure 80%) della loro capacità, in una situazione per cui la chiave di indicizzazione occupa mediamente uno spazio pari al 20% (oppure 15%) del record della relazione.

12. Illustrare vantaggi e svantaggi dei record a lunghezza fissa nella memorizzazione di dati nei database relazionali.

- Nei record a lunghezza fissa come pro ho che l'indirizzamento è semplice: base+offset (lo trovo nel catalogo). Essendo lunghezza fissa posso facilmente stimare quante pagine servono per una query (so quanti record ci sono in una pagina esattamente), e questo mi permette di avere molta più prevedibilità nel sistema. Un altro vantaggio è che avrò molto meno partizionamento rispetto a un record a lunghezza variabile, poiché se elimino un record un altro potrà tranquillamente prendere il suo posto. Il principale problema è che se scelgo una lunghezza troppo grande rischio comunque di avere sprechi di spazi, se troppo piccola invece, potrebbero non starci i dati.

13. Illustrare, anche facendo riferimento ad un esempio, l'algoritmo di linear hashing

- L'idea alla base è tenere sotto controllo l'occupazione globale dell'indice. L'indice viene "riaggiustato" al raggiungimento di una certa soglia di occupazione globale dell'hashing. L'indice usa la funzione hi(k) = k mod N dove k è la key e N è il numero di bucket.
  Supponiamo di avere un hashing su 4 bucket, ognuno può contenere 100 keys (non è realistico), questo indice ha una capienza di 400. Se decido che la capienza massima tollerabile è del 75%, quando complessivamente raggiungo le 300 chiavi eseguo un'operazione di ristrutturazione che si compone in questo modo:
    - Identifica il bucket puntato dallo split pointer
    - Effettua lo split del bucket, ovvero fa in modo che gli elementi inseriti nel bucket vengano distribuiti su due bucket (quello corrente e quello aggiunto alla fine).
    - Ad ogni split:
      - lo split pointer si muove di una posizione in avanti, quando arriva alla fine ricomincia da capo e aggiungo una funzione h con mod 2N
      - si aggiunge una funzione hi+1(K) = k mod 2N
    - Al primo split ho due funzioni e aumentano all'aumentare degli split. Una volta splittati tutti gli N bucket arriverò ad avere una nuova funzione h0 con una nuova N pari a 2N e la procedura riprende dall'inizio.

  Non è efficiente nei casi di indici non uniformi perché se la concentrazione è in un bucket, io lo posso raddoppiare solo una volta per giro creando uno spreco di pagine.
  Lo Split pointer si muove secondo la politica round robin (da sx a dx e poi ri-inizia da capo).
  Non esclude overflow pages, il peso degli overflow non incide in maniera così pesante. Le uso per contenere l'eccesso delle pagine piene prima di avere un'occupazione totale >= a quella tollerata.

14. Quali sono i vantaggi derivanti dall'uso dei record a lunghezza variabile in un file di indice

- L'organizzazione intra-page con record a lunghezza variabile prevede l'utilizzo di una struttura dati ausiliaria: la directory. La directory permette di avere un ulteriore livello di indirizzamento ai record, in quanto ogni slot della directory contiene una coppia <puntatore al record, record_lenght> che permettono di individuare la posizione del record nella pagina. Quindi a fronte di operazioni di inserimento/cancellazione dei record nella pagina quello che viene modificato è il puntatore nella directory piuttosto che il RID del record, di conseguenza tale operazione è trasparente ai riferimenti esterni al record, come possono essere gli indici definiti sulla relazione, che rimangono inalterati. Con un'organizzazione della pagina packed, quindi record a dimensione fissa, a fronte di operazioni di cancellazione, si ha il problema della modifica del RID. Questo accade poichè il RID è una coppia <page_id,slot_id> e per mantenere lo spazio libero in maniera compatta si riorganizzano i record, si modificano gli slot_id e di conseguenza il RID, invalidando i riferimenti esterni.
  Svantaggi: Nei record di lunghezza variabile invece, nel caso "field count" (presenza di un separatore) ho come contro che posso fare solo accesso sequenziale (ma non è troppo problematico, ho tutto in ram). Inoltre, è difficile trovare sequenze di separatori che non siano presenti nei dati, possono essere molto lunghe e sprecare spazio. Invece, nel caso in cui all'inizio del record ho una serie di puntatori che puntano all'inizio di ogni attributo, allora come pro, risparmio spazio rispetto a field count per via dei delimitatori, ma come contro ho un numero limite di campi da poter indirizzare, (ES. se ho puntatori da 1 byte posso avere al massimo 2^8 indirizzi).

15. Abbiamo discusso a lezione l'occupazione delle pagine di memoria da parte di due diverse strutture ad indice gerarchico, ISAM e B+Tree. In particolare, abbiamo sottolineato che ISAM prevede di riempire le pagine dell'indice al 100%, mentre il B+Tree raggiunge tipicamente un'occupazione del 70-80%. Indicare spiegandoli:

a) vantaggi e svantaggi del tasso di occupazione delle pagine da parte di ISAM
b) vantaggi e svantaggi del tasso di occupazione delle pagine nei B+Tree
c) Si consideri un DB in cui non siano siano previste query di range, ma in cui sia possibile avere chiavi duplicate. Preferireste indicizzare con ISAM o con B+Tree? Motivare la risposta

- Pro e contro ISAM: occupo al 100% e quindi occupo memo memoria. Assumo poca dinamicità, e quindi usare le pagine al 100% significa a fronte di ogni query dover leggere meno pagine. Se ho le pagine al 100% sfrutto meglio lo spazio, ho un fan-out maggiore, l'albero ha meno figli. Arrivo ai nodi foglia in modo efficiente, tranne se ho delle situazioni di overflow, in questo caso è inefficiente.

  Pro e contro B+Tree: riempio solo il 70-80 %, siamo in situazione dinamica, e mi rende altamente probabile che a fronte di modifiche, non si scenda sotto al 50% e non si chieda split dei nodi e ristrutturazione degli stessi, ma allo stesso tempo ho sempre indice gerarchico. Riempiendo solo al 70-80, potrebbero essere necessario più spazio per gestire i dati (e quindi più costoso)

  Preferirei Il B+Tree, perchè la struttura dell'ISAM è più statica; quindi, prevede un minor numero di inserimenti e/o cancellazioni possibile, in quanto potrebbero causare pagine di overflow a. Se ho dei duplicati, è preferibile memorizzare i dati all'interno di un B+Tree dove, oltre una certa soglia, non verranno generate pagine di overflow, ma verrà splittata la foglia di interesse continuando a mantenere l'albero bilanciato.

16. Discutere quali sarebbero gli svantaggi derivanti da una bassa utilizzazione delle pagine negli indici di tipo Hash

- L'utilizzazione delle pagine deve essere abbastanza alta perché non posso non usare delle pagine per prevenire overflow, in quanto se ho un file che vuole richiedere delle pagine da portare in memoria, rischio di portarmi in memoria pagine con troppi pochi dati, non avendo quindi il giusto bilanciamento tra la lunghezza delle catene di overflow e una buona utilizzazione delle pagine. Infatti, nelle varie tipologie di hash index, si predilige la velocità di accesso ai bucket (dove troviamo foglie/puntatori a foglia) con un'occupazione media del 25%, piuttosto che alla gestione ottimizzata delle pagine, con svantaggi pesanti che dipendono dal tipo di soluzione implementata per evitare le collisioni:
    - long overflow chain: crescita esponenziale n° bucket e ricostruzione hashing index (critico)
    - extendible-H: sbilanciamento dei dati (molti bucket vuoti a causa dei raddoppi)
    - linear-H: possibile presenza di numerose funzioni di hashing, aumentando il tempo necessario per l'hash finale della chiave

17. Descrivi l'utilità e cosa contiene il System Catalog

- È un file, una componente del dbms che tiene i meta sulla situazione dei file (oltre le statistiche degli accessi, ci sono le informazioni meta sulla struttura dei file). È quella parte dello schema che tiene ad esempio le informazioni sullo schema (come sono gli attributi, com'è indicizzato, lunghezza fissa/variabile, ecc.). Serve ad esempio quando bisogna effettuare una valutazione. Per esempio a fronte di 500 record quanti valori diversi assume, serve per stimare (ad esempio) mediante euristiche le cardinalità dei join.

18. State lavorando per una ditta che produce software per DBMS relazionali e il manager vi chiede un'opinione circa una modifica che intende proporre, per esporre agli utenti finali gli identificatori di record "RowId", per consentire agli utenti di accedere direttamente ai record nel database. Quali sono i pro e i contro di questa feature? Quale sarebbe il vostro suggerimento, in generale? Motivare la risposta.

- Personalmente la reputo una proposta non adeguata, perché come vantaggio (caso remoto) se l'utente finale avesse accesso alla relazione in cui è presente il RowID, potrebbe accedere in modo diretto e univoco al record che gli interessa senza costi aggiuntivi. Ma, di contro, è un'idea costosa sotto tutti i punti di vista: sia per il mantenimento della colonna RowID, che per l'esposizione della struttura interna della tabella e del DB, che rappresenta un potenziale rischio per la sicurezza. Quindi si può fare, ma non se il rowId è una chiave numerica auto incrementale, se è una stringa random allora sì.

19. L'indice ISAM memorizza le pagine di ciascun livello dell'albero in modo contiguo sul disco, mentre i B+Tree usano liste doppiamente linkate. Quali sono le ragioni alla base di questa scelta? Inoltre

a) Illustrate i vantaggi dell'ISAM rispetto ai B+Tree;
b) Illustrare i vantaggi dei B+Tree rispetto all'ISAM.

- ISAM nasce come struttura statica, dove l'interesse è la permanenza del dato piuttosto che la sua modifica, invece un B+Tree ha bisogno, anche a livello delle foglie, di un modo veloce e dinamico per accedere al dato di interesse senza dover fare troppe ricerche. A livello di foglia, un B+Tree utilizza una double linked list per permettere ricerche sequenziali veloci (ad esempio per query di range, dove si trova la foglia interessata e si va avanti fino alla fine del range di ricerca), cosa che ISAM fa già di suo, dato che l'organizzazione su disco è tale da essere già clustered. Nel B+Tree le pagine a livello di foglia sono linkate anche perché potrebbero essere su posizioni di disco totalmente o parzialmente differenti. In ISAM i dati vengono memorizzati in maniera ordinata e

contigua mentre nei B+- Tree possono essere memorizzati anche in maniera non ordinata e contigua
Vantaggi B+-Tree, a differenza di ISAM: tutti i primi nodi interni che la memoria centrale (RAM) può contenere vengono mantenuti su di essa mentre il resto dei nodi e le foglie vengono lasciate su memoria di massa (disco). Ciò permette una maggior velocità di ricerca. Inoltre non c'è la possibilità di generare pagine di overflow (cosa che negli ISAM succede). Inoltre nell'ISAM ogni pagina può essere ordinata secondo un unico criterio di ordinamento (cosa che nel B+Tree viene meno)
Vantaggi degli ISAM rispetto al B+Tree: è il fatto che ISAM può avere un albero meno profondo rispetto a B+Tree, garantendo una migliore efficienza. Favorisce accessi concorrenti, il locking è meno pesante sull'albero, ma comunque non occupa più memoria dei B+Tree

20. È possibile definire un hash-index come clustered?
(simile) È possibile avere un indice basato su hashing che sia un indice "clustered"? Motivare la risposta

- Indice clustered vuol dire che l'ordine delle chiavi è uguale a quello dei puntatori, per definizione l'hashing non ha le data-entry ordinate e quindi non è previsto un particolare ordinamento (se lo sono è un caso particolare che non è previsto dalla definizione di hashing). Pertanto, non è possibile avere un indice hashing clustered a causa della natura stessa della funzione di hash, i diversi bucket devono essere organizzati in modo da non essere clustered, altrimenti stiamo usando una funzione di tipo seriale e non avrei benefici prestazionali di accessi più rapidi. Un indice clustered rimane clustered per tutta la sua durata vitale. Inoltre, se sappiamo che il file è ordinato, possiamo raggiungere mediante l'indice hashing il primo record e proseguire la lettura del file in maniera sequenziale ma anche questa non è una situazione prevista dalla definizione dell'indice clustered (infatti è un caso molto raro).

21. Supponendo di avere un indice di tipo B+tree su dei dati, e supponendo che questo indice non renda efficienti le query, quali potrebbero essere le cause? Come risolverle? (almeno due)
(simile) Supponiamo che abbiate implementato una applicazione per database che faccia pesante affidamento su indici di tipo B+ tree per migliorare i tempi di risposta delle query. E supponiamo che nonostante l'uso di questi indici la valutazione delle query richieda ancora troppo tempo, e vi sia chiesto di migliorare ulteriormente i tempi di risposta delle query. Indicare almeno 2 di quelle che secondo voi possono essere le ragioni alla base dei problemi osservati, e discutere quali soluzioni proporreste per affrontare tali problemi.

- Le cause possono essere:
  1. chiavi troppo grosse e quindi fanout troppo basso.
  2. indice uncluster con tanti duplicati → causa riletture di pagine multiple
  3. riempimento dell'albero troppo alto e quindi continue ristrutturazioni
  4. query che richiedono molte scansioni complete

  Per risolvere si potrebbe migliorare aumentare il fan-out dell'albero. Un maggior fattore di diramazione rende l'albero meno profondo e le foglie, in teoria, sono più velocemente raggiungibili. Se si facessero molte query di tipo equality search rispetto a quelle di range, si potrebbe pensare di utilizzare un hash index, al posto del B+tree. Usando un indice cluster si evita di rileggere pagine inutili. Utilizzando chiavi di ricerca più selettive oppure chiavi di ricerca complesse.

22. Descrivere il tempo per l'accesso ai dati (Disco).

- l tempo per accedere ai dati si calcola sommando il tempo di seek, il ritardo della rotazione e il tempo di trasferimento.

Buffer Manager

23. Illustrare gli algoritmi basati su Working Set e su Hot Set per la gestione del buffer.

- Working Set: prevede che la priorità di una pagina dovrebbe essere determinata sulla base della frequenza rispetto a cui viene utilizzata la relazione, di cui questa pagina fa parte. Se ho una relazione memorizzata in 5 pagine diverse (alcune delle quali sono già presenti nel buffer) a queste pagine associo una certa priorità e la politica di rimpiazzamento terrà conto di questa priorità, in particolare, candiderà ad essere rimpiazzate, le pagine con priorità più bassa. La priorità è la frequenza con la quale la relazione che contiene quella pagina viene utilizzata (se ho presenti nel buffer un certo numero di pagine provenienti dalla stessa relazione, queste hanno tutte la stessa priorità). L'idea di fondo è che non uso genericamente LRU o MRU ma associo alle pagine che non sono di indice, una priorità, che è associata alla frequenza che cattura la frequenza d'uso della relazione di cui le pagine fanno parte". Ci sono alcune limitazioni, si usa l'MRU per evitare il sequential floading, ma non è sempre la scelta migliore, non c'è una "personalizzazione" rispetto all'operatore che si sta implementando.
  Hot Set: questo algoritmo assegna alla singola query il numero di frame di buffer pari al hotpoint della query, ovvero il più piccolo numero di frame allocabili che garantisce il drastico calo di page fault. Ogni query, dunque, ha il suo buffer, ma al contrario delle relazioni non si conosce a priori il numero di query diverse. L'hotpoint può variare per ogni operatore, per esempio su una selezione potremo avere un hotpoint pari a 1 (leggo e decido se mandare in output record oppure no, ma sovrascrivo sempre lo stesso frame del buffer), mentre su un Join un hotpoint più cospicuo (ad esempio se la relazione interna fosse grande 30 pagine potrei avere un hotpoint grande 30 e leggere tutta la relazione interna in un solo paso). Il problema di questo algoritmo è che siccome la stima dei frame è basata sul caso peggiore, tende ad essere "memory hungry" quindi può sovrastimare il numero di pagine necessarie e dunque sprecare spazio.

24. Illustrare l'algoritmo per la gestione del buffer basato sull'idea di working set (quello che va sotto il nome di "new" algoritm), mettendone in evidenza vantaggi e svantaggi.

- L'algoritmo associa a ogni relazione un certo numero di frame di buffer in base alla sua priorità (frequenza con cui la relazione viene usata). Le relazioni, quindi, vengono messe in una coda con priorità ordinata. Quando una query ha bisogno di frame di buffer per delle pagine, se ce ne sono di disponibili le concede, altrimenti è necessario un rimpiazzamento. In questo algoritmo, ogni transazione opera seguendo la sua politica di rimpiazzamento sulla porzione di buffer che gli è stata assegnata, e tipicamente si sceglie di usare MRU, per evitare il sequential flooding.
  Svantaggi**:** Lavorando su diverse relazioni che possono essere anche molto diverse non abbiamo una politica di rimpiazzamento migliore da scegliere e quindi si possono creare diversi problemi. Andando ad utilizzare MRU potrebbe essere controproducente in alcuni contesti dinamici, dove ci sono poche query frequenti. Inoltre, il mantenimento dell'ordine della coda è costoso.
  Vantaggi: Con questo algoritmo l'ordinamento non è costoso perché avviene sulle relazioni, che in genere sono poche o comunque minori rispetto al numero di frame nel buffer. Inoltre, più la pagina è utilizzata più rimarrà in cima, quindi i frame delle relazioni minori verranno scartati e in caso di sequential flooding non si intaccheranno i frame più usati.

25. Quali sono le differenze nei costi di LRU e dell'algoritmo del working set in merito alla gestione della coda delle frequenze?

    Il working set è un algoritmo che lavora su relazioni gestendo una coda organizzata per frequenza e puó utilizzare LRU o MRU, LRU è una politica di rimpiazzamento che segue il principio di localitá temporale e libera il frame che non viene utilizzato da piú tempo. Se dovessimo basarci sulla frequenza utilizzeremo il LFU che elimina il frame meno utilizzato(utilizzato meno di frequente).

26. Abbiamo visto che l'algoritmo Hotset per la gestione del buffer, che analizza il comportamento ciclico delle operazioni di join e assegna a ciascuna query un pool di buffer di dimensione pari al suo Hotset, finisce spesso per allocare spazio di buffer eccessivo rispetto alle effettive necessità. Illustrare le ragioni alla base di questo fenomeno.

- La ragione per il quale si ha un'allocazione eccessiva è dovuta al fatto che il buffer manager fa una stima sulla quantità di buffer necessaria ad evitare un elevato numero di page fault per ogni query. Quindi, può sovrastimare il numero di pagine necessarie e dunque sprecare spazio. Perché la dimensione dell'offset viene stimata in base alle informazioni storage nel catalogo, e le stime quindi sono pessimistiche (per essere sicuri di allocare sufficienti risorse), per cui occupiamo più spazio (sprecandolo). Nel caso del join, per esempio, ci possono essere dei record che vengono utilizzati più volte e quindi può succedere che venga allocato spazio di buffer eccessivo. Invece, una sottostima, determinerebbe un numero di page fault maggiore che sarebbe più svantaggioso di avere spazio sprecato; quindi, si preferisce l'allocazione di maggiori frame per la query.

27. Fan Out, come funziona e si calcola il fattore di ramificazione? È relativo a tutto l'albero oppure solamente al nodo con più archi?

- Il Fan-out: È dato dal rapporto tra la dimensione della pagina di disco e la dimensione dell'informazione associata alle singole chiavi di ricerca. Quindi è un fattore di ramificazione e d è relativo solo al numero di puntatori presenti nel nodo.

28. Come si calcola la profondità dell'albero? Dimensione delle pagine / dimensione delle foglie?

- (Riassunto Sapino MAADB.pdf) È dato dal logaritmo fattore di ramificazione del numero di punti da indicizzare. Ad ogni livello il numero di nodi che vengo a trovare lo ottengo di dividendo quelli di livello più basso più il fattore di ramificazione. Passando da un livello all'altro ottengo un'operazione di tipo logaritmico. La profondità dell'albero, quindi, può essere stimata come logk(n) dove k è il massimo numero di puntatori ai figli che possono stare in una pagina e n è il numero delle foglie. K è uguale alla dimensione di una pagina diviso la dimensione di una chiave di ricerca; dunque, è bene avere chiavi di ricerca piccole di modo da avere un k più grande.

29. Abbiamo discusso a lezione il fatto che le strategie di rimpiazzamento delle pagine nel buffer basate sul principio di località temporale (quali ad esempio LRU) potrebbero rivelarsi inefficienti nel caso delle basi di dati relazionali. Illustrare le ragioni alla base di questa affermazione, anche facendo riferimento ad un esempio. Quali pattern di accesso beneficiano nella politica di rimpiazzamento LRU?

- Perché in caso si presentino dei cicli una politica come quella LRU rischierebbe di causare sequential floating e sarebbe quindi meglio una politica MRU.Ad esempio avendo a disposizione 6 frame di buffer e una relazione di 8 pagine, se si utilizzasse LRU i primi 6 frame verrebbero caricati senza problemi e letti ma per le letture successive si avrebbe per ogni nuova lettura un page fault. I pattern di accesso che beneficiano della politica LRU sono per esempio Straight sequential e clustered sequential poiché non essendo presenti cicli ma dovendo scorrere il file non avviene il problema del sequential flooding.

30. Abbiamo visto che nella gestione del buffer l'algoritmo del clock viene usato in alternativa all'LRU per ragioni di efficienza. Abbiamo anche visto che uno dei problemi che possono derivare dall'uso della politica LRU è il fenomeno del sequential flooding che può emergere in presenza di determinati pattern di accesso. Discutere se l'algoritmo del clock potrebbe anche risolvere questo problema.
(simile) L'algoritmo del clock (ovvero LRU euristico) risolve il sequential flooding?

(simile) Nella gestione del buffer, la politica LRU a fronte di alcuni pattern di accesso dà luogo al Sequential Flooding. Lo stesso fenomeno si può verificare anche nel caso del Clock?

- No, l'algoritmo del clock è una metaeuristica che approssima il comportamento del LRU. Il clock viene utilizzato esclusivamente per ridurre l'overhead dovuto al mantenimento della coda di priorità (quindi dell'ordinamento) tra i frame del buffer per identificare quello meno utilizzato di recente. L'algoritmo CLOCK, così come LRU ed LFU, è basato sul principio di località temporale, il cui rappresenta il motivo principale che porta al problema del sequential flooding. Questo comportamento degenere si presenta nei casi di pattern di accesso 'loop' (ciclici). Una policy di replacement MRU è quello che lo risolve definitivamente.

Operatori

31. Si consideri l'algoritmo di Block Nested Loop Join. Assumendo di dover operare su due relazioni, R ed S, costituite rispettivamente da *r* ed *s* pagine, dire quali altre informazioni sono necessarie per poter stimare costo di questa operazione di join. (Non limitatevi ad un elenco, ma spiegate Il motivo per cui le informazioni che Indicate sono necessari).

- Non essendo un'operazione simmetrica il costo è funzione di chi è la relazione interna e chi quella esterna, per cui:
    - Devo decidere quale delle due è l'interna e quale è l'esterna, una che guida il ciclo e quella su cui si cicla. Dato un numero B + 1 (+1 per l'output) di pagine del buffer. Posso tenere: un numero di pagine minore di B – 1 per la relazione esterna (quindi B – 2 o meno ancora); le rimanenti pagine per la relazione interna; un frame per l'output
    - Cardinalità di R e S, che ci permetterebbe di sapere se affidare più frame del buffer a una relazione rispetto che all'altra
    - La selettività degli attributi, quante coppie multiple ci sono?
    - Come distribuisco i blocchi alle due relazioni?
    - Le relazioni sono ordinate? (Potrei ottimizzare grazie ad un pre-sorted).

32. Siano date due relazioni R ed S . Si supponga che R contenga 100.000 record a lunghezza fissa, ciascuno di 50 bytes ed S contenga 150.000 record a lunghezza fissa, ciascuno di 100 bytes. Si supponga inoltre che la dimensione della pagina disco sia di 50.000 bytes. Qual è il costo della valutazione dell'operazione di join tra R ed S (R join S) secondo l'algoritmo di Nested Loop Join? E secondo il Sort Merge Join? Motivare la risposta (riportando anche i risultati intermedi dei calcoli), ed evidenziare eventuali ulteriori informazioni aggiuntive che potrebbero essere utilizzate per scegliere l'una o l'altra implementazione dell'operazione di Join.

- Come informazioni aggiuntive da evidenziare è importante sapere se le relazioni sono già ordinate, se le chiavi del join sono tutte univoche o ci sono duplicati? (se ci sono duplicati il costo del Join

aumenta, perché su alcune pagine devo ciclare per leggere più coppie con quell'attributo), infine sapere quante pagine di buffer ho a disposizione.
Per quanto riguarda il calcolo dei costi: costi page oriented nested loop join: conviene che la relazione esterna sia la più piccola, quindi R. R ha 100 pagine con 1000 record ognuna. S ha 300 pagine con 500 record ognuna. Il costo è: M + M * N = 100 + 100 * 300 = 30100.
Il costo sort merge ottimizzato: costo lineare: 3(M + N) = 3*(400) = 1200
Il costo sort merge non ottimizzato: N log N + M log M + M+N = 200 + 750 + 400 = 1350 al caso migliore. caso peggiore: N log N + M log M + M*N = 200 + 750 + 30000 = 30950 (questo dipende se ci sono duplicati o meno nelle condizioni di join).

Il simple nested join è l'algoritmo di join più semplice e versatile e rispetto al sort merge join non è bloccante. Il sort merge join migliora notevolmente i risultati poiché il costo è nettamente inferiore rispetto al simple ma ha il problema di essere bloccante, ovvero dover aspettare che le relazioni vengano ordinate a causa di questo la soluzione ottima è il sort merge join migliorato che una volta eseguita la runs eseguo il join. Un'informazione utile che potremmo utilizzare è sapere il numero di frame del buffer che abbiamo a disposizione, poiché questo ci permetterebbe di ridurre il numero di runs dell'ordinamento.

33. Qual è il costo della valutazione dell'operazione di join tra R ed S (R join S) secondo l'algoritmo di Nested Loop Join? E secondo il Sort Merge Join? Motivare la risposta (riportando anche i risultati intermedi dei calcoli), ed evidenziare eventuali ulteriori informazioni aggiuntive che potrebbero essere utilizzate per scegliere l'una o l'altra implementazione dell'operazione di Join

- Supponendo che le relazioni abbiamo rispettivamente M pagine per R e N pagine per S, il costo del Nested loop join e di M + M.r * M * N avente R come relazione esterna e S come relazione interna, la relazione interna va scelta in base alla cardinalità poiché una relazione interna più piccola porta vantaggi a livello di computazione. per il caso sort merge join il costo si attesta a M log M + N log N + M + N nel caso migliore e M log M + N log N + M*N nel caso peggiore tutto questo nel caso della versione standard dell'algoritmo. la versione da me presentata ha come principale svantaggio il fatto di essere bloccante e quindi dover aspettare ordinamento del file. Non c'è un scelta migliore, in caso si voglia un algoritmo più versatile, semplice e soprattutto non bloccante si sceglie il nested loop join altrimenti si sceglie il sort merge join. Segnalo inoltre che il sort merge join a livello di costi e tendenzialmente da preferire.

34. Descrivere quali pattern d'accesso vengono utilizzati nell'algoritmo sort-merge-join, e come vengono allocati i frame del buffer secondo l'algoritmo DB-min in questo caso.
(simile) Quali pattern d'accesso sono coinvolti nell'algoritmo di Sort Merge Join che abbiamo discusso? Descrivere le modalità secondo cui l'algoritmo DBMin allocherebbe i frame buffer per questo operatore.

- I pattern d'accesso sono straight sequential e clustered sequential. Straight sequential in quanto io scansiono entrambe le relazioni durante la fase di partizionamento per comparare le tuple della prima relazione all'interno di una partizione, con le tuple della seconda relazione, all interno della seconda relazione. Clustered sequential in quanto in quanto dopo la fase di ordinamento, vado a partizionare i valori delle tuple delle relazioni in base all attributo di join.
Per quanto riguarda db-min in sostanza si applica LRU per il clustered e MRU per lo straight sequential (per evitare sequential flooding).

35. Quali pattern di accesso sono coinvolti nell'algoritmo di Block Nested Loop Join? In quale modo l'algoritmo DBMin allocherebbe e gestirebbe i frame di buffer per questo operatore?

- Per il Block Nested Loop Join usiamo i pattern looping sequential e hierarchical sequential (se l'attributo di join ha un indice) ma dipende da quanti frame del buffer abbiamo. In caso la relazione interna possa essere mantenuta tutta nei frame potremmo usare dei pattern sequential. In ogni caso DBmin calcolerà i locality set della query e capirà il pattern di accesso migliore in base al tipo di file e le cardinalitá delle relazioni.

36. Abbiamo discusso diverse implementazioni dell'operatore di join, notando che l'index nested loop join e il block nested loop join possono essere gli algoritmi da preferirsi, in situazioni diverse. Dovendo valutare una query con condizioni di join ad alta selettività, quale tra le due alternative sceglieresti?

- Dipende dal tipo di struttura a indice di cui disponiamo, se avessimo un indice cluster con chiave di ricerca di tipo 3 potremo sfruttare il fatto che nelle foglie troviamo tutti i record ordinati a cui siamo interessati e quindi ridurre il costo. In caso di un indice un cluster con chiavi di ricerca di tipo 3 rileggiamo alcune pagine diverse volte, ma eviteremo probabilmente molte pagine inutili. per l'utilizzo del block nested loop join il tutto dipende da quanti frame del buffer abbiamo a disposizione; se infatti il numero di frame sarà uguale alla cardinalità di pagine di una delle due relazioni questo potrebbe risparmiarci diverso tempo per la ricerca.

37. È possibile migliorare le prestazioni (ridurre il costo) allocando più buffer all'esecuzione dell'algoritmo di sort-merge join?
(simile) Si possono migliorare le prestazioni del sort-merge join allocando più buffer durante l'esecuzione?

- Si, è possibile. È necessario precisare che non è tanto la quantità dei frame di buffer a disposizione quanto invece la predisposizione delle tabelle per un buon merge-join. L'algoritmo di sort-merge join si articola in due fasi: 1) sorting, 2) merging. In particolare, nella fase 1) per ordinare le due relazioni viene utilizzato l'algoritmo di external sorting. Abbiamo visto che un'implementazione naive di quest'ultimo si basa sull'utilizzo di 3 frame del buffer, 2 per l'input e uno per l'output con un costo totale di 2N*(log_2(N)+1). Utilizzando B frame del buffer (B-1 per l'input e 1 per l'output) è possibile ridurre drasticamente il costo complessivo, portandolo a 2N*(log_B-1(N/B)+1) osservando come si riduce il numero di runs da N ad N/B e il numero di passi da Log_2(n)+1 a Log_B-1(N/B)+1. Quindi con il fatto che elementi presenti in più pagine della relazione debbano essere messi in join con altrettanti elementi, avere più buffer è positivo. Perciò, nella fase di sort, avere il buffer più grande mi permette di fare run più lunghi (quindi vado a ridurre il numero di passaggi per raggiungere il file ordinato), mentre nella fase di merge mi permette di evitare di caricare più volte le stesse pagine.

38. Abbiamo visto un algoritmo di external sort che applica le idee e le proprietà alla base del merge-sort. Perché scegliere proprio il merge-sort? quali difficoltà si incontrerebbero se si scegliesse di definire un "external - quicksort"?

- Il fatto è che il quicksort fa tanti scambi nell'array di elementi da ordinare, fare tutti questi scambi su disco comporta un numero di operazioni altissimo. Bisogna perciò cercare di limitare il numero di operazioni ed il merge-sort fa questo: crea dei run più lunghi possibili da portare in memoria (quindi run lunghi = numero più basso di run = numero piu basso di iterazioni i/o) e poi fa il sort in memoria.
  Infatti, il quicksort lavora in-place mentre il mergesort no. Questa caratteristica del lavorare in-place del quicksort lo rende molto inefficente per gli External Sort (perché per ogni confronto deve

portare pagine in memoria), ovvero gli algoritmi di ordinamento che lavorano in memoria secondaria (portando in memoria primaria solo alcune pagine e non tutte).
Quindi in algoritmi come il Merge Sort, si divide sempre la relazione a metà e via via si fondono porzioni ordinate di relazioni andando a generare porzioni di relazioni ordinate di larghezza doppia. La natura degli accessi che si fanno sui file di queste dimensioni è tale da rendere più conveniente lo sfruttamento dei dati. Il merge-sort richiede dello spazio aggiuntivo ma con opportuni accorgimenti, permette di sfruttare bene il buffer.

39. Se il sorted-merge join è più conveniente del nested join soprattutto quando le relazioni sono già ordinate, perché dovremmo usare il nested? qual è il vantaggio del nested?

- sorted-merge join fa parte della categoria delle operazioni bloccanti mente il nested loop join non lo è. No blocking vuol dire: man mano che i risultati arrivano dall'operazione che sta generando l'input il join può iniziare, ad esempio se devo fare un join di 1000 tuple con una relazione di 1500 tuple appena ho materiale sufficiente per riempire le pagine di buffer possono iniziare a lavorare. Con il sorted-merge join invece devo rimanere bloccato fino a quando tutti i miei operandi sono stati generati. Con il no blocking ci potrebbe essere un piccolo ritardo prima di ottenere il risultato. Quindi fin tanto che l'input non è ordinato nel suo complesso (non è completamente conosciuto) non è possibile applicare le operazioni di sort-merge join.

40. Quali sono alcune tecniche per l'operazione di eliminazione dei duplicati in una relazione? (simile) Illustrare 2 metodi alternativi per l'operatore di eliminazione dei duplicati nelle basi di dati relazionali.

- Per l'eliminazione dei duplicati si potrebbe: Avere un indice sulla relazione, sull'attributo di cui si vogliono trovare duplicati e allora si trovano subito i duplicati scandendo l'indice ordinato. Es: l'Indice di hash in quanto se trovassi più valori nello stesso bucket avrei i valori "collidenti" nelle pagine di overflow corrispondenti. Oppure con sorting ed eliminazione, ma è fondamentale il costo: se la relazione è già ordinata il costo è lineare altrimenti il costo è quadratico e per ogni elemento bisogna controllare a seguire se ci sono dei duplicati oppure si precede questa operazione da un ordinamento ma in questo caso il costo lo si trasferisce sul sort (quindi diventa costosa non per la ricerca dei duplicati ma per l'ordinamento che bisogna fare come prerequisito per cercare in maniera lineare il duplicato).
  Un altro modo per eliminare i duplicati è mettere la relazione in join con se stessa, se si fa R join R ciascun record verrà associato a tutti gli altri in cui è presente la stessa chiave quindi nel join risultante appariranno le occorrenze dei valori e da quelli si potrà facilmente eliminare i duplicati ma il join resta costoso.

41. L'eliminazione dei duplicati nei DB relazionali è un'operazione costosa. Discutere se secondo voi sarebbe preferibile utilizzare un indice b+tree unclustered o un indice b+tree clustered per realizzare l'eliminazione dei duplicati, motivando la scelta proposta.

- Fra i due ha più senso un B+Tree Clustered con chiavi di tipo 3 perchè mi trovo tutti i valori nella stessa pagina (o in quelle successive) e quindi contengo i costi di lettura

42. Come funziona il block nested? Se ho B blocchi, quante pagine bisogna leggere? (è richiesto quindi istanziare la formula sul caso specifico, è importante motivare la risposta)

- L'algoritmo block nested loop join è un algoritmo di join che cerca di sfruttare al meglio il numero di frame messi a disposizione dal buffer, dedicando più frame per l'esecuzione. Non c'è un caso

standard ma dipende dal tipo di relazioni e la loro cardinalità, se la relazione interna è abbastanza piccola da essere contenuta tutta nei frame del buffer allora affideremmo tutti i frame a lei (con costo di M+(N/B)*M), nel caso inverso(la relazione esterna piccola abbastanza per essere contenuta nei frame) faremo il contrario (con costo di M/B + N*(M/2)). Se le due relazioni sono di cardinalità simile divideremo i frame equamente tra le due relazioni.

43. Si consideri l'algoritmo di Sort Merge Join discusso a lezione. Si assuma di avere due relazioni R ed S, rispettivamente di N ed M pagine. Quali altre informazioni sarebbero necessarie per stimare il costo di questa operazione di join? (Non limitatevi ad un elenco, ma motivate la risposta).

- È necessario sapere:
    - le cardinalità delle due relazioni
    - Se i file sono ordinati oppure no, se il file non è ordinato e usassimo la versione ottimizzata del sort merge join sarebbe importante sapere quante pagine del buffer abbiamo a disposizione per stimare il costo, e per poter fare delle Run in parallelo (oppure con solo due run posso avere tutto quello che mi serve già in memoria)
    - se ci sono tanti valori duplicati nelle relazioni, se sì, il costo del Join aumenta, perché su alcune pagine devo ciclare per leggere più coppie con quell'attributo. Il costo da additivo diventa moltiplicativo.

44. Se sto considerando il join e devo associare un'implementazione scegliendo il sort-merge join, come sarà l'albero?

- Simmetrico perchè il sort-merge join non risente dell'ordinamento, con altri join invece il costo ne risente e cambia a seconda di quale relazione è interna / esterna.

Ottimizzazione delle query

45. Enunciare e discutere almeno 3 ragioni alla base della scelta di non cercare il piano OTTIMO nella fase di ottimizzazione delle query

- Di seguito 3 ragioni per cui è preferibile non cercare il piano ottimo:
    1. Il costo per trovare l'ottimo potrebbe essere superiore rispetto al costo che si andrebbe a risparmiare applicando il piano ottimo rispetto a un sub-ottimo
    2. il calcolo dei costi si basa su stime a causa del fatto che il System Catalog non viene mantenuto costantemente aggiornato a causa dei costi di I/O e del carico elevato di operazioni su DB.
    3. le stime tendono a essere pessimistiche

46. La tecnica basata sulla programmazione dinamica per la scelta dell'ordine di valutazione delle operazioni di join presuppone che le operazioni di proiezione e selezioni siano anticipate (ossia spinte verso le foglie del query tree) il più possibile per ridurre le cardinalità e le dimensioni dei record che devono essere combinati dal join. Abbiamo visto che anche l'eliminazione dei duplicati può essere utilizzata come strategia per la riduzione del numero dei record. Ritenete che sarebbe opportuno spingere verso le foglie anche l'eliminazione dei duplicati, oltre alla selezione e alla proiezione? Motivare la risposta.

- Ritengo che spingere verso le foglie del query tree l'operatore di eliminazione dei duplicati sia una mossa inefficiente dal punto di vista della complessità computazionale in quanto è un'operazione ausiliaria ed è tra le più costose perché richiede fare prima sort oppure un join a seconda

dell'algoritmo. Pertanto, dobbiamo diminuire la complessità e metterla prima della selezione e proiezione aumenterebbe la complessità che è in funzione della cardinalità dell'input. Inoltre, non si dovrebbe anticipare prima del join poiché causerebbe una possibile perdita di informazioni per il join.

47. Abbiamo visto a lezione che l'ottimizzatore delle query basata sulla programmazione dinamica presuppone che i piani di valutazione delle query siano "left-deep". Sarebbe possibile applicare ottimizzazione delle query basata sulla programmazione dinamica per piani di valutazione "right-deep"? Motivare la risposta

- È possibile applicare un algoritmo di ottimizzazione sul right-deep (infatti posso fare operazioni basate su indici o anche su cicli) ma, essendo bloccante, le ottimizzazioni sarebbero limitate e, sempre poiché bloccante, sfrutto l'indice solo sul primo livello dell'albero senza avere la possibilità di procedere al livello successivo. Per questo motivo l'ottimizzatore preferisce i left-deep rispetto ai right-deep

48. I Query Optimizer cercano di evitare i piani peggiori anziché cercare di individuare i piani ottimi. Discutere le ragioni alla base di questa scelta.

- Perché con i piani peggiori, può essere che vengano portate in memoria un numero elevato di pagine. L'ottimizzazione fa l'esatto opposto: passa dallo statement dichiarativo a uno statement procedurale (capire cosa fa una query) cercando di considerare la query con costo minore. La query meno costosa è quella il cui risultato occupa un numero di pagine in memoria relativamente piccolo. (confrontato con altre sotto-query). Quindi si cerca di evitare di trovare piani peggiori al posto di trovare i piani ottimi perché ci metteremo troppo a trovare l'ottimo, e le statistiche stimate sono approssimate ed imprecise; quindi, non avremmo neanche le informazioni necessarie ad avere un ottimo.

49. Data la query: *"Select * FROM DIPENDENTI WHERE Anno_di_Assunzione = 2020 AND Salario_annuale > 50.000"*. Discutere se (e in quali condizioni) la valutazione della query beneficerebbe di un indice su "Anno_di Assunzione", su "Salario_annuale", su "<Anno_di Assunzione, Salario_annuale>" o su "<Salario_annuale, Anno_di Assunzione>".

- Data la query, ci sono tre casi principali:
    1. Se disponessimo di un indice su un solo attributo tra "Anno_di_Assunzione" e "Salario_annuale", potremo sfruttarlo per selezionare i valori che matchano la prima condizione di selezione e da questi ci basterebbe escludere i valori che non matchano con la seconda condizione di selezione. Ad esempio, potremo selezionare tutti i record che matchano con Anno_di_Assunzione = 2020 ed eliminare quelli che non rispettano Salario_annuale > 50.000 oppure il viceversa. (filtraggio)
    2. Se disponessimo di un indice su entrambi gli attributi, potremo invece calcolare prima gli insiemi di record ID corrispondenti a entrambi i criteri di selezione (2 ricerche) e poi calcolare l'intersezione insiemistica tra questi due result set. Però in questo caso, per fare ciò avremo bisogno di due indici unclustered.(intersezione insiemistica)
    3. Con indice con chiave composita potrei ordinare i record per <Anno, Salario> andando a scandire sequenzialmente tutti i record fino a che la condizione su salario è valida. (chiave composita) Si beneficerebbe soprattutto di un indice composito (quindi un indice posto su entrambi) su anno_di_assunzione in quanto in questa maniera mi posizionerei subito sull'attributo anno_di_assunzione = 2020 e di seguito troverei tutti i dipendenti assunti

nell'anno di mio interesse e mi basterebbe applicare la query di range sul salario_annuale e alla prima voce > 50.000 mi tirerei giù tutti i valori fino all'ultimo.

Diciamo quindi che per quel che concerne la stima finale del costo, sicuramente la soluzione (2) è la più costosa, in quanto stiamo effettuando i nostri calcoli su due indici di tipo unclustered. Ma non solo, nello scegliere l'approccio tra (1) e (2) dovremo considerare anche la selettività del predicato, (in quanto un predicato più selettivo risulta più economico nella stima finale dei costi) e se la relazione in questione è ordinata oppure no (se lo fosse, il costo si abbasserebbe). La soluzione (3) è la più efficiente ma se fosse ordinata al contrario (<Salario,Anno>) ciò non è più vero. Inoltre, sempre nella soluzione (3) ad indice composito conviene usarlo solo se si vuole usare l'interrogazione tante volte.

50. Si considerino due diversi algoritmi di ordinamento in-memory, l'uno molto veloce, l'altro più lento ma in grado di creare run più lunghi. Quale usereste come component di un algoritmo di ordinamento esterno? Motivare la risposta.

- Per algoritmi di ordinamento esterno dobbiamo partire da dei run ordinati, e quelli li facciamo con algoritmi in memory. Quando si confronta la velocità degli algoritmi in memory, si confronta la velocità a un ordine di grandezza inferiore rispetto all'ordine di grandezza degli algoritmi non in ram. Per cui è da privilegiare il costo dell'algoritmo più lento (in quanto non dobbiamo pensare di velocizzare un costo in memoria) ma solo se questo mi fa dei run più lunghi (perché ne beneficio nella fusione delle run in ram, risparmio tempo lì). Ogni fusione comporta la lettura e riscrittura di tutto il file, e determina accessi a disco. Meno volte passo per il file, meglio è perché è costoso.

51. Si supponga di voler ordinare un file che occupa 500 pagine utilizzando l'algoritmo di External Merge sort. Assumendo di avere a disposizione 6 pagine nel buffer, calcolare il costo totale del l'operazione di ordinamento.

- Avendo B=6 pagine del buffer, posso usare la versione orientata al blocco. Posso usare 6 pagine per volta per creare runs lunghi 6, dopodiché di queste 6 posso usarne una per scaricare l'output e 5 per portarmi in memoria 5 run per volta, per fare in modo ottimizzato i merge.
  Run iniziali = 500 / 6 = 83,333 ~ 84 run
  Li ho costruiti con costo 2 (uno per leggere e uno per scrivere) * 500 = 1000.
  Quante fusioni devo fare? $log_5 84$ (logaritmo in base 5 di 84 in limite superiore).
  Costo totale = 2*500 *[1 + $log_5 84$] con il logaritmo arrotondato per eccesso.

52. Illustrare mediante un esempio il principio di sub-query optimality su cui si basa l'ottimizzatore delle query relazionali.

- Il principio che regola il comportamento dell'ottimizzazione è il principio di ottimalità delle sottoquery, dove, per ottimizzare a livello globale una query composta da un certo numero di sottoquery, io posso considerare individualmente i costi delle diverse sottoquery e quando io sommo questi costi, sceglierò come piano globale, quello che risulta da quelli locali che complessivamente hanno costi minori. Quindi, il piano ottimale per una query complessiva è quello costituito a partire da piano sub-ottimali per le sottoquery.. Infatti, un piano ottimo è quello che ha i sotto piani ottimi (si può usare la ricorsione).
  Per esempio, per stimare il costo del join R1 ⋈ R2 ⋈ R3 ⋈ R4 io dovrò calcolare il costo del join di R1 ⋈ R2 ⋈ R3, il costo del join di R1 ⋈ R3 ⋈ R4 ecc.. Allo stesso modo per valutare R1 ⋈ R2 ⋈ R3 dovrò valutare R1 ⋈ R2 ; R2 ⋈ R3...ecc. Diciamo che con la procedura ricorsiva mi trovo a

valutare più volte lo stesso costo, infatti la soluzione sarebbe quella di utilizzare la programmazione dinamica, sia procedendo top-down che bottom-up (più frequente).

53. Per l'ottimizzazione delle query basata sui costi è necessario disporre di statistiche relative alla distribuzione dei dati. Si consideri la seguente query: SELECT DISTINCT r.NAME,s.NAME,t.NAME FROM R r, S s, T t WHERE r.AGE>=s.AGE AND r.AGE<=t.AGE. Proporre tre diversi query plans per questa query e dire (spiegandone le ragioni) quali statistiche sarebbero necessarie scegliere il piano da utilizzarsi tra i tre proposti

- I 3 query plans possono essere:
    a) R >< S >< T
    b) S >< R >< T
    c) R >< T >< S

  Il costo dell'operazione è in funzione della cardinalità degli operandi: dobbiamo anticipare l'operazione che riduce questa cardinalità. Ci conviene perciò anticipare quello che è soddisfatto da poche, dovendo poi confrontare meno elementi nelle operazioni successive. Dobbiamo conoscere la distribuzione dei valori (min/max e come sono distribuiti). Dobbiamo anche conoscere se ho un indice sull'attributo età, numero valori distinct per età, per considerare certe implementazioni utilizzate.

54. Perché ci serve l'ottimizzazione delle query?

- Il linguaggio SQL è un linguaggio dichiarativo, per eseguire le varie operazioni bisogna tradurlo in un linguaggio procedurale. Il processo si articola in: Parsing e traduzione da SQL ad operatori di algebra relazionale (piani logici); Scelta delle implementazioni degli operatori (piano fisico); Valutazione dei costi dei piani fisici per scegliere quella computazionalmente più efficiente.

55. Qual è il vantaggio dei query plan "left-deep" rispetto a quelli "right-deep"? Giustificare la risposta

- i left deep mi consentono di utilizzare "pesantemente gli indici"; ho implementazioni non blocking perchè per far partire l'algoritmo devo solamente aver bisogno che sia completa la relazione di destra per ogni relazione di sinistra. In questo modo ho un maggiore margine di ottimizzazione. Per quanto riguarda i right deep, l'indice si può sfruttare solo nella foglie più profonda a destra dell'albero, e inoltre ho un algoritmo di blocking in cui, per l'implementazione degli algoritmi, la relazione interna (ovvero quella di destra) deve essere completamente disponibile.

56. Sia R(A, B, C) una relazione in un DB, e sia SELECT * FROM R WHERE (A >1000) AND (A <2000) AND (B >1000) AND (B <2000) una query molto frequente in questo database. Assumendo che si stia cercando di decidere se creare un indice B-tree basato sulla chiave composta A:B, o sulla chiave B:A, illustrare i criteri che si devono prendere in considerazione per compiere una scelta ragionata tra le due alternative.

- Una chiave composta su range potrebbe essere non ottimale. In questo caso abbiamo un range sia su A che su B e si dovrebbe anticipare l'attributo più selettivo. Se assumiamo che una delle due abbiano una cardinalità inferiore e una selettività alta questi forse sarebbero dei vantaggi sufficienti per scegliere di ordinare prima su quella chiave.

57. Abbiamo visto in classe che "random independent" è uno dei pattern di accesso comunemente presenti nelle basi di dati. Proporre e spiegare un esempio di operazione di query processing che richieda spesso pattern di accesso “random independent.

- Un esempio è una query che richieda accesso a dei dati con indice, come un Join su un attributo con indice. In questo caso, per ogni valore della relazione esterna va a verificare se il valore c'è nella relazione interna. (es: cerco un nome di persona). Quindi arrivo alle foglie tramite l'indice, accedo alla relazione interna tramite il pattern random indipendent (perché avendo tutti “accessi indipendenti” sulla relazione interna posso fare quindi in maniera randomica). Mentre sulla relazione esterna è sequential, perché scandisco la relazione una riga alla volta.

58. Discuti la seguente ipotesi di lavoro: il costo del risultato finale (costo della query) sia la somma dei costi di tutti i nodi

- Non è necessariamente sempre realistica perché implementazioni diverse possono lasciare l’output meno costoso per implementazioni successive alternative. L’unico aspetto che mi interessa stimare per calcolare i costi è la cardinalità del risultato prodotto dall’applicazione di ogni operatore.

59. Alcuni tipi di query che richiedono dati da relazioni indicizzate possono essere valutate senza accedere a relazioni del database, limitandosi all’analisi dell’indice (query index-only). Proporre un esempio di query che possa essere valutata in modalità index-only (indicando la query proposta e l’indice/gli indici di cui si presuppone l’esistenza), e un esempio di query sulle stesse relazioni, e con gli stessi indici, che non possa essere valutata in modalità index-only. (fatta alla simulazione)

- Una query index only è: *SELECT cognome FROM utenti WHERE cognome> naretto and cognome<zara.*
  Per essere una query index only dobbiamo avere un indice costruito su cognome. Un esempio di una query non index only è: *SELECT * FROM utenti WHERE cognome> naretto and cognome<zara*

Gestione delle transazioni

60. Si considerino due DBMS che applichino diversi meccanismi per il controllo della concorrenza, applicando rispettivamente Strict 2PL e Optimistic 2PL. Quale dei due necessita di più spazio di memoria? Motivare la risposta.

- Ritengo che tra i due meccanismi pur lavorando meglio Optimistic 2LP è anche quello che occupa più spazio perché bisogna mantenere copie di dati in ram per diverso tempo quando si vogliono fare operazioni di scrittura. Questo perché lo strict 2PL non si rilascia il lock su un oggetto fino al commit della transazione, acqusisce uno shared lock per leggere e un un exclusive lock per scrivere. I lock vengono rilasciati solo quando la transazione che possedeva il lock termina.

61. Si considera un Transaction Manager che utilizzi il protocollo 2PL anziché lo strict 2PL. Discutere quale sarebbe l’impatto di questa scelta sul crash manager che si basa sul WAL per il crash recovery.

- Il protocollo 2PL può causare una cascata di abort, a differenza dello strict 2PL che garantiva l’esecuzione delle transazioni in isolamento in quanto nel 2PL il lock acquisito su un oggetto veniva rilasciato solo in fase di COMMIT e quindi nessuno poteva accedere concorrentemente agli oggetti sui cui avevo acquisito il lock.

62. Proporre un esempio di scheduling di transazioni non serializzabili, discutendo le ragioni della non serializzabilità

- Non è Serializzabile in quanto non confing equivalent, questo perché se si genera un conflitto non è nello stesso ordine di uno schedule seriale.
  T1: R(A)W(A) R(B)W(B)
  T2: R(A)W(A)R(B)W(B)

63. Abbiamo visto che il Concurrency Control ottimistico elimina la necessità di ricorrere al lock, utilizzando in alterativa copie private degli oggetti e TimeStamps. illustrare, anche facendo ferimento a un esempio, come questi metodi garantiscano la conflict serializability senza ricorrere ad alcun lock.

- Un esempio è l'Algoritmo di Kung-Robinson. Quando una transazione è attivata può iniziare subito a lavorare senza prendere lock. Abbiamo una prima fase di read dove si legge un oggetto, se ne crea una copia privata (ovvero non si condividono le pagine nel buffer). La fase di Validate dove si verifica l'esistenza di cicli sui conflitti (se ci sono cicli la transazione abortisce). E la fase di write che rende pubbliche le modifiche apportate alla copia privata, se la fase precedente è andata a buon fine. Avremo uno spreco di memoria nel buffer, ma meno overhead tra una lettura/scrittura e l'altra (come nel caso dei lock).
  A ciascuna transazione, è associato un timestamp quando inizia la fase di validazione. E si assume che tutte le transazioni che hanno iniziato la validazione siano conflict serializable, per esempio, quando arriva una transazione nella fase di validazione, essa assume che tutte le altre transazioni che erano in validazione prima di lei fossero conflict serializable. La transazione controlla di essere conflict serializable rispetto alle transazioni con timestamp precedente, e se non lo è abortisce (dunque non vengono killate le transazioni più vecchie ma le più giovani).

64. Esponi i pro e contro degli approcci basati su timestamp e lock

- I lock hanno un approccio più pessimistico che blocca le transazioni poiché tendono a serializzare le operazioni, questo può causare il problema dei deadlock ma riduce il rischio di abort in cascata (dipenda anche dal protocollo utilizzato). I timestamp invece hanno un approccio più ottimistico che evita di mettere in attesa/bloccare altre transazioni favorendo un sistema di priorità dato dal timestamp stesso, evita di causare deadlock ma rischia di causare abort in cascata.

65. È vero che il lettore non abortisce mai perché riesce sempre a trovare una copia?

- No, il lettore può trovarsi nella situazione di abortire ma per via di aborti in cascata e non per una situazione sua (conflict WR)

66. Dato uno schedule, cosa vuol dire che è Conflict Serializable? Come riconoscerlo?

- Uno schedule conflict Serializable è un schedule appartenente al sottoinsieme dello schedule serializzable, la sua caratteristica e avere interliving e in presenza di conflitti le transazioni appaiono ordinate nello stesso modo di uno schedule seriale

67. Perché i lock permettono la prevenzione di cicli?

- L'acquisizione del lock vincola nel grafo delle dipendenze un'unica direzionalità degli archi alla volta, quello nella direzione opposta non potrà essere inserito fino a quando non si fa l'unlock precedente rimuovendo il primo arco. In sostanza, l'utilizzo dei lock forza la serializzabilità

68. Discutere le motivazioni e le principali idee alla base del multiresolution locking (Multiple-Granularity). Perché serve e come viene realizzato?

- Serve perché abbiamo bisogno di gestire i lock a livelli di granularità diversa (db → tabella → pagina → tupla/colonna → cella). La soluzione sono i Multiple-Granularity Locks, infatti in questa tecnica esiste un'altra categoria di lock detta intention lock. Una transazione che intenda accedere in lettura a un oggetto di granularità larga (per esempio vuole leggere tutta una tabella), allora dichiara uno shared lock sull'oggetto (per esempio tabella) di interesse. Se una transazione vuole scrivere su un oggetto di granularità larga, allora dichiara un lock esclusivo su quell'oggetto. Se l'intenzione di una transazione è modificare una tabella (per esempio) ma andando a modificarne un singolo record, il lock esclusivo lo si vuole acquisire solo su quel record, tuttavia, prima di acquisire il lock esclusivo sul record, la transazione deve acquisire l'intention lock sui livelli superiori (minor granularità), in questo caso si acquisisce intention lock su pagina, tabella e database. Infatti, un intention lock su un oggetto X indica la volontà di agire su un oggetto Y che è contenuto all'interno di X ovvero che ha granularità più fine di X (gli intention lock possono essere di scrittura o lettura). Quando si intende acquisire un lock su un oggetto, si parte dalla radice della gerarchia (db) e si comincia ad acquisire intention lock dal livello più alto scendendo fino al livello che ci interessa (escluso), raggiunto il livello dell'oggetto desiderato si richiede un lock normale (non intention). Infine il rilascio del lock invece avviene in maniera bottom up (così non ho problemi con le transazioni in sospeso sui lock).

69. Quali sono le idee principali alla base dell'algoritmo di Kung-Robinson per il controllo della concorrenza delle transazioni ottimistico? In quali casi può essere preferibile questo algoritmo di concorrenza rispetto a quelli pessimistici?

- L'idea principale sta nel fatto che quando si vuole andare a fare una modifica di un dato, ne creo una copia locale quando eseguo la read, dopodiché modifico la copia del file e eseguo la fase di validazione, dove si effettuano i test per controllare che non ci siano conflitti, e infine scrivo. Generalmente preferiamo un approccio di tipo ottimistico quando sappiamo che la maggior parte delle transazioni non sono conflittuali (magari guardando la storia delle transazioni e conflitti dal system catalogo), oppure quando ci possiamo permettere di spendere meno tempo a controllare la serializzabilità e quindi risparmiare risorse preziose. Questo algoritmo è ottimo dei casi in cui si accede spesso a dei dati poiché se usassimo un algoritmo basato su lock rallentiamo molto la parallelizzazione.

70. Abbiamo visto che l'approccio basato su "serial validation" per la gestione del controllo della concorrenza ottimistico elimina la necessità di ricorrere al lock, sostituendone l'uso mediante copie private degli oggetti, time stamps e l'uso di porzioni di codice in sezioni critiche. Non essendo lock, ovviamente non si presentano casi di deadlock e di conseguenza non si presentano aborti di transazioni per risolvere i casi di deadlock. Date queste considerazioni, possiamo concludere che il controllo della concorrenza ottimistico basato su "serial validation" è completamente privo di situazioni di aborto di transazioni? Motivare la risposta.
(simile) La serial validation elimina la necessita di usare il lock. non essendoci lock non c'e neanche deadlock. quindi non possono verificarsi casi di aborto per risolvere i deadlock. possiamo dire che la serial validation è privo di situazioni di aborto?

- No, non siamo privi di situazioni di aborto, a prescindere dalla serial validation. L'utente può sempre causare un aborto volontario, o magari anche in cascata. La serial validation non utilizza i lock ma i timestamp, questo permette di evitare i deadlock, ma non è l'unica ragione per causare un aborto di una transazione. Infatti, un utente è la causa primaria degli aborti su transazioni, molto spesso. Uno dei problemi principali legati alla serial validation è la situazione di Starvation (fallimenti ripetuti da parte della stessa transazione), se ci si trova in tale situazione la soluzione è quella di procedere con l'abort della transazione. Prima di procedere con il restart della transazione

(con un nuovo timestamp) si cerca di individuare e risolvere le cause che mi portano ad avere una situazione di Starvation.

71. Spiegare qual è il ruolo dei timestamp nel consentire schedule serializzabili?

- Associando un timestamp a ogni transazione siamo in grado di prevenire il deadlock perché diamo una priorità alle transazioni e quando si tenta di entrare in attesa si creano solo archi in attesa che vanno in un'unica direzione. Abbiamo due politiche: la Wait-die e la Wound-wait che usano la priorità del timestamp

Crash Recovery

72. Discutere il ruolo della transaction table e della dirty-page table nella gestione delle situazioni di crash. Per quali ragioni vengono mantenute in RAM e non su disco?

- Entrambe le tabelle sono cruciali nell'algoritmo per il crash recovery. La transaction table conserva una entry per ciascuna transazione attiva. Ogni entry conserva l'id della transazione, il suo stato (running/committed/aborted) e il lastLSN ovvero il suo più recente LSN. La dirty page table tiene traccia delle pagine dirty, e per ciascuna pagina dirty c'è il recLSN, ovvero l'LSN del primo log record che ha reso dirty la pagina. Si va a salvare il primo LSN e non l'ultimo (perchè l'ultimo LSN lo ho già all'interno della pagina come ho scritto prima). Sono quindi mantenute in ram perché dovrei modificarle troppo di frequente, e sarebbe troppo costo portare su disco ogni aggiornamento, io cerco di salvare su disco solo quello necessario; quindi, voglio evitare di salvare dati superflui.

73. Quali sarebbero i problemi causati dall'inversione della fase di redo e della fase di undo nell'algoritmo di crash recovery?
(simile) Quali problemi si andrebbe incontro se l'algoritmo di crash recovery invertisse l'ordine di esecuzione delle due fasi di undo e di redo?

- Se facessi undo prima di redo, il rischio sarà che i redo potrebbero re-introdurre modifiche già disfatte (che non devono persistere). Infatti, si fa prima la redo e poi undo perché prima devo rifare tutte le operazioni, poi eventualmente dopo disfo quelle delle transazioni che hanno fatto abort. In questo modo mantengo consistenza perché seguo il naturale ordine delle operazioni: eseguo delle operazioni e dopo eventualmente abortisco. Potrei complicare la fase di redo, per evitare che si chieda di rifare delle azioni che successivamente sarebbero da disfare e nel complesso non dovrebbero essere rifatte, ma rimane una scelta comunque rischiosa: quando si disfa qualcosa, si vuole registrarne il corrispondente compensation log record (CLR), usarlo in caso di crash successivo, e tale gestione ibrida potrebbe complicare ulteriormente anche la gestione del log per tener traccia di info utili in caso di crash a seguire.

74. Si consideri una sequenza di log entries (log i, log i+1, ..., log j) che erano in ram nel momento in cui si è verificato un crash, sono andate perse insieme ai corrispondenti update effettuati dalla transizione. Discutere se questa perdita causa o non causa problemi nel crash recovery

- No, perché se sono in RAM significa che riguardano cambiamenti che non sono stati memorizzati su disco (altrimenti il WAL scrivendo un record di log successivo avrebbe salvato anche i precedenti compresi quelli) e quindi non hanno sporcato il disco. È vero che i log verranno persi, però verranno

perse anche le relative operazioni e sarebbe come se non fossero mai avvenute quindi senza causare problemi

75. Una delle ragioni per cui sono necessari algoritmi per la gestione del recovery basati sul log è legata alla presenza delle operazioni di scrittura/lettura su disco. Con la diffusione dei nuovi e più veloci dispositivi di memorizzazione, quali i dischi basati su tecnologia flash, gli update su disco sarebbero più economici. Discutere se sarebbero ancora necessari algoritmi per la gestione del crash recovery log based qualora i dischi diventassero così veloci da rendere estremamente efficienti le operazioni di aggiornamento su disco.

- I due problemi sono indipendenti l'uno dall'altro. Il crash non è relativo al fatto che sto scrivendo ed è crashato il sistema, ma il sistema è crashato con write su disco e su ram, e io devo riportare il sistema in una situazione consistente, indipendentemente dalla tecnologia del disco. Quello che potrebbe cambiare è che si potrebbe non ritardare più le scritture su disco (applicare una politica force quindi). Però gli UNDO saranno sempre da gestire in caso di utilizzo di approccio force/steal. Se io portassi su disco dirty pages delle relazioni non ancora committed, se ho un crash devo rispristinare la situazione pulita in memoria, per questo serve il log. Con le operazioni più efficienti avrei operazioni di write, ma non avrò mai garanzia che su disco avrò una situazione consistente.

76. Perché nel Write Ahead Logging, è meno costoso scrivere il log su disco rispetto a scrivere le pagine di dati modificati?
(simile) Per quale motivo è opportuno salvare su disco il log invece che le pagine modificate?

- Principalmente perché nel log ci sono più informazioni, poi ciascuna transazione opera potenzialmente su più pagine e tante scritture su pagine diverse comportano tanti costi read/write e tempi di seek. Inoltre, evito un alto numero di accessi casuali al disco, perché appendo tutto al fondo del log in modo sequenziale, siccome il Log File è un file "append-only", il che lo rende molto più efficiente rispetto ad accedere ai dati contenuti nel disco (seek time + rotation time + transfer time). Inoltre, posso scrivere su disco in modo "compatto" per cui è la strategia più efficiente. La scrittura di N modifiche sul log, in genere, coinvolge la scrittura di una sola o poche pagine, mentre scrivere N modifiche potenzialmente coinvolge N pagine diverse.

77. Che differenza c'è tra Abort e Crash?

- Abort: Potenzialmente voluto dall'utente, ma in ogni caso non si ha perdita di dati, poiché ho ancora tutti i dati presenti nel buffer, si è semplicemente deciso di cancellare quello che stavo facendo per tornare indietro. Nella gestione dell'abort, posso contare sulle entry del log file aggiornate (cosa è stato fatto, perché, da chi e dove) e sui vari metadati, perché non è mancata la corrente.
  Crash: Situazione improvvisa di emergenza nella quale manca la corrente o accade un errore non gestito e si ha perdita di dati in ram. Tutto quello che era nel buffer è perduto. Per gestire questa situazione posso usare l'algoritmo di crash recovery a tre fasi:
  1. Analisi: in questa fase ricostruisco le Dirty Page Table e la Transaction Table, che mi servono per procedere nelle fasi successive.
  2. Redo: scorrendo il log file ricostruisco i dati com'erano prima (comprese le operazioni di cui fare undo)
  3. Undo: scorrendo il log file annullo le operazioni che devono essere annullate, segnandole nel file di log con dei CLR

78. Perché l’abort è più facilmente gestibile rispetto a un crash di sistema?

- Nel caso dell’abort abbiamo tutte le informazioni in ram e quindi non dobbiamo ricostruire, la Transaction Table e la Dirty Page Table non vengono perse. L’aborto è più facile perché è controllato, fosse mancata la corrente avrei perso le informazioni in memoria principale che avrei dovuto ricostruire. È vero che un abort è più facilmente gestibile rispetto a un crash ma è anche vero che potrebbe succedere un crash durante un aborto.

79. Come posso garantire che nel log ci sia tutto ciò che mi serve per poter ripristinare la situazione consistente e gestire transazioni committed / non committed?

- Mi basta usare WAL, una procedura progettata per consentire perdite di transazioni non committed (in questo caso, il log che era ancora nel buffer è andato perso), ma che a fronte di un crash, mi garantirà che il log salvato su disco sarà aggiornato fino all'ultimo commit e fino all'ultima scrittura di pagina su disco. Inoltre la presenza di log record speciali come End e Compensation log records (CLRs) aiutano il transaction manager a ricostruire la situazione iniziale consistente tramite l’algoritmo trifase (analisi,redo,undo).