Domande Esame

1. Vantaggi nell'utilizzo di record a lunghezza variabile nell indice
    a. vantaggi: per un indice e' importante maximizar il fanout delle index entries, un indice a lunghezza variabile permette di farlo in maniera ottimale. stesso discorso nelle data entries di un indice, non ho bisogno di calcolare la lunghezza massima di un record.
2. L'algoritmo del clock (ovvero LRU euristico) risolve il sequential flooding?
    a. No, dato che approssima LRU ed LRU soffre del sequential flooding
3. Si possono migliorare le prestazioni del sort-merge join allocando piu buffer durante l'esecuzione?
    a. si, nella fase di sort, buffer più grande mi permette di fare run più lunghi. nella fase di merge mi permette di evitare di caricare più volte le stesse pagine.
4. select distinct r.name, s.name, t.name from r,s,t where r.age >= s.age and r.age <= t.age. Proporre 3 diversi query plans per la query e dire quali statistiche sono necessarie per scegliere il migliore dei query plans.
    a. RxSxT; SxRxT; RxTxS
    b. cardinalità delle relazioni, numero valori distinct per age
5. 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 e' privo di situazioni di aborto?
    a. No, perche l utente puo mettere aborti programmati

1. Spiegare qual è il ruolo dei timestamp nel consentire schedule serializzabili?
    a. vedi appunti.
2. 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.
    a. No, visto che non sono state scritte su disco non influenzeranno il recovery (?) → esatto perche non c'e' stato commit se non sono su disco(si usa WAL)
3. 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 di questa scelta? inoltre: illustrare i vantaggi di ISAM rispetto i B+ tree e i vantaggi dei B+ tree rispetto ISAM.
    a. scritto qua sotto negli appunti
4. Si considera l algoritmo di sort merge join. Si assuma di avere due relazioni R e S di rispettivamente N e M pagine. Quali altri info sarebbero necessarie per stimare il costo di questa operazione di join?
    a. quantità di pagine di buffer a disposizione, presenza di duplicati, se le relazioni sono già ordinate
5. Si considerino 2 diversi alg di ordinamento in run. uno molto veloce e l'altro lento ma in grado di creare run più lunghi. quali usereste come component di un algo di ordinamento esterno?
    a. è meglio quello con run più lunghi perchè si risparmia letture su disco

6. si supponga di voler ordinare un file che occupa 500 pagine utilizzando external merge sort. assumendo di avere a disposizione 6 pagine di buffer calcolare il costo totale dell'operazione di ordinamento.
    a. run iniziali = 500 / 6 = 84 (approssimato per eccesso) costo= 2x500 (per ognuno una lettura e scrittura) costo fusioni= log5 84 costo tot = la + di questi due(vedi appunti sotto)
7. "random independent" è uno dei pattern di accesso comunemente presente nei dbms. proporre e spiegare un esempio di operazione di query processing che richieda spesso pattern di accesso "random independent"
    a. query su indici unclusterd o basati hash
8. abbiamo visto che l algo di 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 a l suo hotset, finisce spesso per allocare spazio di buffer eccessivo rispetto alle effettive necessità. illustrare le ragioni alla base di questo fenomeno
    a. succede perché la stima dell hotset è stimata pessimisticamente per essere sicuri di allocare sufficienti risorse ma si spreca sulla fine
9. una delle ragioni per cui sono necessari algo 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 algo per la gestione del crash recovery log based qualora i dischi diventassero così veloci da rendere estremamente efficienti le operazioni di aggiornamento su disco
    a. i due problemi sono indipendenti l uno dall altro. 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
10. perché i dbms relazionali sono più efficienti di quelli ad oggetti per le ottimizzazioni e esecuzioni di query?
    a. essendo più vicino al livello fisico si possono effettuare diverse ottimizzazioni (es pagine contigue,) discorso prevedibilità, pochi operatori dell'algebra relazionale...
11. block nested loop join: due relazioni R e S costituite da r e s pagine dire quali altre info sono necessarie per potere stimare il costo di questa operazione di join?
    a. il costo va in base a quale relazione è esterna e quale interna. quanto è grande il blocco. come distribuisco i blocchi alle due relazioni. sapere se sono ordinate
12. spiegare i vantaggi e svantaggi di isam e b+tree del tasso di occupazione delle pagine da parte di entrambe. con non una query di range e chiavi duplicate cosa è meglio usare? (stessa domanda 1 della prova prima)
13. 5. 6. ha foto gabbo

ALTRE DOMANDE

1. Discutere quali sarebbero gli svantaggi derivanti da un bassa utilizzazione delle pagine negli indici di tipo Hash
    a. questo può accadere se ho una funzione di hashing con un rapporto 1:1 sulle chiavi. questo porta ad avere un file di hashing troppo grande. Seconda cosa rischio di non riempire a pieno le pagine: se ogni bucket e' una pagina, e se in ogni bucket posso mettere 10 record, però ho una funzione di hash troppo precisa che mi porta ad usare poco i bucket, rischio di avere ad esempio pagine riempite da un solo elemento → usage al 10%. Quando poi devo fare una scan completa faccio troppo passate di pagine per leggere tutti i dati. Quindi di solito negli indici di hash si rendono possibili le collisioni (chiavi diverse nello stesso bucket), a patto che i dati siano distribuiti in maniera uniforme.
2. 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.
    a. si e' da fare ma non se il rowId e' una chiave numerica auto incrementale. se e' una stringa random OK. I vantaggi e che accedi direttamente al record in maniera univoca.
3. 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>".
    a. l'indice sarebbe da fare sull'attributo piu selettivo. Quindi il match dell'anno e' sicuramente più selettivo che un range su stipendio. Avere le foglie ordinate anche su stipendio porta sicuramente vantaggio perché basta una scansione sequenziale una volta arrivata alla foglia <2020, 50000>. Quindi la risposta per me e' <Anno_di Assunzione, Salario_annuale>
4. Discutere le motivazioni e le principali idee alla base del multiresolution locking. Perché serve e come viene realizzato?
    a. leggersi questo
5. 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"?
    a. quicksort fa tanti scambi nell'array di elementi da ordinare, fare tutti questi scambi su disco comporta un numero di operazioni altissimo. bisogna quindi cercare di limitare il numero di operazione 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.
6. 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.

    a. vantaggi: assegna priorità per ogni relazione in base al suo utilizzo (più frame per relazioni più usate)
    b. svantaggi: assegnare la priorità alla relazione non cattura il metodo di accesso alla relazione stessa (se uso LRU per una certa relazione e poi ci accedo a quella relazione con un looping sequential sono un coglione → ovviamente perché non posso determinare a priori come ci accedo)
7. Perché per alcuni tipi di dati non sono adatti database relazionali?
    a. 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, dove le relazioni tra nodi vengono salvate fisicamente e di conseguenza accedere un nodo in relazione con un altro ha costo costante. inoltre 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 di campi o relazioni sui singoli nodi/documenti. al contrario dei database relazioni dove c'e' uno schema da seguire e tutti i dati devono seguire quello schema.
8. È possibile definire un hash-index come clustered?
    a. no, hash non ordina
9. 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)
    a. chiavi troppo grosse ergo fanout troppo basso
    b. indice fatto su chiavi poco consone ad essere indice
10. 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.
    a. buffer-manager → query optimizer: in base alle pagine che ho a disposizione nel buffer, il query optimizer puo adottare un approccio rispetto ad un altro. viceversa in base al tipo di query, il buffer gestisce le pagine in maniera diversa
    b. transaction manager → recovery manager: il transaction manager gestisce i file di log usando una certa politica (wal o altre). il crash recovery sulla base di questa informazione opera in un certo modo.
    c. lock manager → transaction manager: sulla base delle tipologie di lock gestisco le transazioni in modi differenti.
    d. buffer manager → recovery manager: se uso steal/nosteal o force/noforce il crash recovery deve fare certe cose o meno.
    e. lock manager → recovery manager: se ho strict due pl ho undo piu easy
11. Quali sarebbero i problemi causati dall'inversione della fase di redo e della fase di undo nell'algoritmo di crash recovery?
    a. secondo me 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. Se faccio prima gli undo e poi i redo, non ha senso perché sto cercando di disfare cose che non ho ancora fatto. inoltre fare i redo dopo gli undo ripeto del lavoro già fatto visto che redo funziona anche per i record di tipo crl.

12. Quali sono alcune tecniche per l'operazione di eliminazione dei duplicati in una relazione?
    a. sort then delete
    b. hashing, se due chiavi ritornano lo stesso valore della funzione di hash → duplicato
13. 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?
    a. perché 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
14. Per quali ragioni in un indice ISAM, le pagine di ciascun livello dell'albero sono memorizzate sul disco in moda contiguo, mentre in un i B-Tree si usano liste doppiamente linkate?
    a. perché isam e' un indice clustered statico con le pagine piene sempre al 100% (con gestione overflow su una parte di file separata), mentre i b-tree sono dinamici con pagine piene al 70% circa. e dato che le foglie non sono allocate in modo contiguo come isam, per poter fare una scansione sequenziale delle foglie senza accedere ogni volta all'albero mi servono i puntatori

15. 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 e' 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.
    a. informazioni aggiuntive utili: le relazioni sono già ordinate? le chiavi del join sono tutte univoche o ci sono duplicati? quante pagine di buffer ho?
    b. 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. costo e' M + M * N = 100 + 100 * 300 = 30100
    c. costo sort merge ottimizzato: costo lineare: 3(M + N) = 3*(400) = 1200
    d. costo sort merge non ott: nlogn + mlogm + M+N = 200 + 750 + 400 = 1350 al caso migliore. caso peggiore: nlogn + mlogm + M*N = 200 + 750 + 30000 = 30950 (questo dipende se ci sono duplicati o meno nella condizioni di join)
16. Illustrare mediante un esempio il principio di sub-query optimality su cui si basa l'ottimizzatore delle query relazionali.
    a. guarda qui
17. 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?
    a. assegno un timestamp per ogni transazione. ogni transazione ha tre stati di lavoro: read, validate, write: leggo i dati, faccio modifiche su copie private,

controllo che le modifiche sia consistenti, se si rendo persistente. uso i timestamp durante la fase di validazione per controllare eventuali conflitti. Insieme ai timestamp tengo conto dei writeset e readset per ogni transazione e devo controllare le intersezioni di questi set fra transazioni per verificare conflitti. in generale approcci ottimistici portano a maggiore concorrenza nel caso i conflitti siano poco frequenti rispetto ad approcci pessimistici. questo perché i pessimistici bloccano con i lock le transazioni a priori(anche quando ce possibilità di nessuna inconsistenza) mentre quelli ottimistici bloccano solamente a posteriori nel caso di inconsistenza.

18. 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 Illustrate i vantaggi dell'ISAM rispetto ai B+Tree Illustrare i vantaggi dei B+Tree rispetto all'ISAM.
    a. ISAM rende più pesante l'eventuale query di range, per via delle presenza delle pagine di overflow, perché devo tener presente che quelle non sono ordinate come gli indici. NON occupa più memoria dei B+Tree. CONTRO: Posso avere un solo indice ISAM per ogni relazione, perché le foglie possono essere ordinate rispetto a un solo attributo. Indice ISAM favorisce accessi concorrenti, modifiche solo a livello di nodo e modifiche non distribuite, il locking è meno pesante. ISAM può essere ristrutturato (per via delle pagine di overflow) SAPINO: ISAM è pensato per situazioni statiche, B+Tree contro: occupa più memoria (occupazione al 70%), più complicato gestire controllo degli accessi (modifiche si possono propagare), non necessita di Lock essendo che l'albero è statico (il lock si mette solo sulle foglie)
19. 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 informazoni sarebbero necessarie per stimare il costo di questa operazione di join? **Non limitatevi ad un elenco, ma motivate la risposta**.
    a. - Sapere se relazioni già ordinate oppure no, perché sennò devo ordinarle io.
       - 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.
       - Quante pagine del buffer a disposizione, per poter fare delle Run in parallelo
20. 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.
    a. SAPINO: Per algoritmi di ordinamento esterno dobbiamo partire da dei runs ordinati, e quelli li facciamo con algoritmi in memory. Quando io confronto la velocità degli algoritmi in memory, confronto 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, ma 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. Risposta: "userei il secondo perché a run più lunghi corrispondono una grossa riduzione degli accessi a disco sarebbe maggiore rispetto all'operazione di ordinamento in RAM"

21. Si supponga di voler ordinare un file che occupa 500 pagine utilizzando l'algoritmo di "extemal merge sort. Assumendo di avere a disposizione 6 pagine nel buffer, calcolare il costo totale dell'operazione di ordinamento.
    a. Avendo 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? $\lceil log_{5}(84) \rceil$ (logaritmo in base 5 di 84 in limite superiore). Costo totale = 2*500 + $\lceil log_{5}(84) \rceil$
22. Abbiamo visto in classe che "random independent" è uno de[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.
    a. 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 Marco, Antonio, ecc.). arrivo alle foglie tramite l'indice. Accesso alla relazione interna tramite il pattern random indipendent, mentre sulla relazione esterna è sequential, perché scandisco la relazione una riga alla volta.
23. 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.
    a. Perché la dimensione dell'offsett viene stimata in base alle informazioni storate nel catalogo, e le stime sono pessimistiche (per essere sicuri di allocare sufficienti risorse), per cui occupiamo più spazio (e si spreca).
24. Una delle ragioni per cui sono necessari algoritmi per la gestione del recovery basati sul log è legata alla presenza delle operazioni di scrittura su/lettura da disco.
    a. Il crash non è relativo al fatto che sto scrivendo ed è crashato il sistema, ma il sistema è crashato con cose scritte su disco e su ram, e io devo riportare il sistema in una situazione consistente, indipendentemente dalla tecnologia el disco. Quello che potrebbe cambiare, se avessi la scrittura super efficiente, è che potrei non ritardare più le scritture su disco. Potrei applicare una politica force. Se io portassi su disco dirty pages delle relazioni non ancora committed, se ho un crash devo rispistinare la situazione pulita in memoria, per questo serve il log.Forse con le operazioni più efficienti avrei meno cose da fare (scrivere), ma non avrò mai garanzia che su disco avrò una situazione consistente.
25. 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 ottimizzabilì 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 affermato inizialmente. Indicare due ragioni a supporto dell'affermazione iniziale dandone spiegazione.
    a. Prevedibilità ed SQL ha meno operatori con un numero limitato di implementazioni e quindi c'è una casistica di ottimizzazioni limitata. Sapino:

L'organizzazione tabulare della relazione è molto vicina all'organizzazione dei record nel disco. Cosa non vera nei DMBS ad oggetti, perché data la presenza di puntatori diventa difficile fare stime ed ottimizzare di conseguenza. Il modello ad oggetti si usa di solito per allocazione continua infatti. Inoltre, essendo l'SQL linguaggio dichiarativo, di domanda all'ottimizzatore l'interrogazione, e lui sceglierà l'implementazione più ottimizzata per quella query.

26. Si consideri l'algorítmo 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 infomazioni 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**.
    a. “è un'operazione simmetrica?” Attenzione a leggere il testo, ho detto che devo fare BNLJ operando su R ed S, quindi NON è simmetrica, e quindi il costo è funzione di chi è la relazione interna e chi quella esterna. L'esterna guida il ciclo e l'interna cicla, Per cui - Devo decidere quale delle due è l'interna e quale è l'esterna - Quanto è grande il blocco? - Cardinalità delle pagine - La selettività degli attributi, quante coppie multiple ci sono - Extra: se le relazioni sono ordinate (se voglio ottimizzare)
27. 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'occupazioen del 70-80%. Indicare spiegangoli:
    - vantaggi e svantaggi del tasso di occupazione delle pagine da parte di ISAM
    - vantaggi e svantaggi del tasso di occupazione delle pagine nei B+Tree
    - 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
    a. Pro e contro ISAM: occupo al 100% e quindi occupo meno 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.
    b. Su B+3 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)
    c. Dipende dalla situazione. Se sono in situazione statica con chiavi duplicate fin dall'inizio, meglio ISAM, perché presuppone memorizzazione contigua e ordinata. Nel B+Tree non è garantito (non necessariamente clustered). Se la situazione fosse molto dinamica sarebbe sconsigliato l'ISAM, perché le chiavi duplicate potrebbero essere non contigue.
28. I Query Optimizer cercano di evitare i piani peggiori anziché cercare di individuare i piani ottimi. Discutere le ragioni alla base di questa scelta.
    a. Perché cerchiamo di evitare piani peggiori, ma non cerchiamo piani ottimi? Perché tanto 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.

29. Abbiamo visto che il Concurrency Control ottimistico elimina la necessità di ricorrere al lock, utilizzando in alternativa copie private degli oggetti e TimeStamps. illustrare, anche facendo riferimento a un esempio, come questi metodi garantiscano la conflict serializability senza ricorrere ad alcun lock.
    a. L'idea fondamentale alla base è “condizione necessaria sufficiente per la conflict serializability è che non ci siano cicli nel grafo delle transazioni“. Queste tecniche garantiscono che (storia degli archi…), inoltre si consentono soltanto conflitti per cui la transazione che agisce per seconda sull'oggetto è più giovane di quella che ha già agito(quindi freccie da Ti a Tj). Si garantisce che ci sia sempre lo stesso ordinamento temporale associato alle diverse transazioni che intervengono nel conflitto. Anziché acquisire Lock per evitare situazioni di dipendenza, non si pongono vincoli a priori, laddove si creano situazioni di conflitto, ammettiamo dipendenze in un solo ordine che ammettiamo.

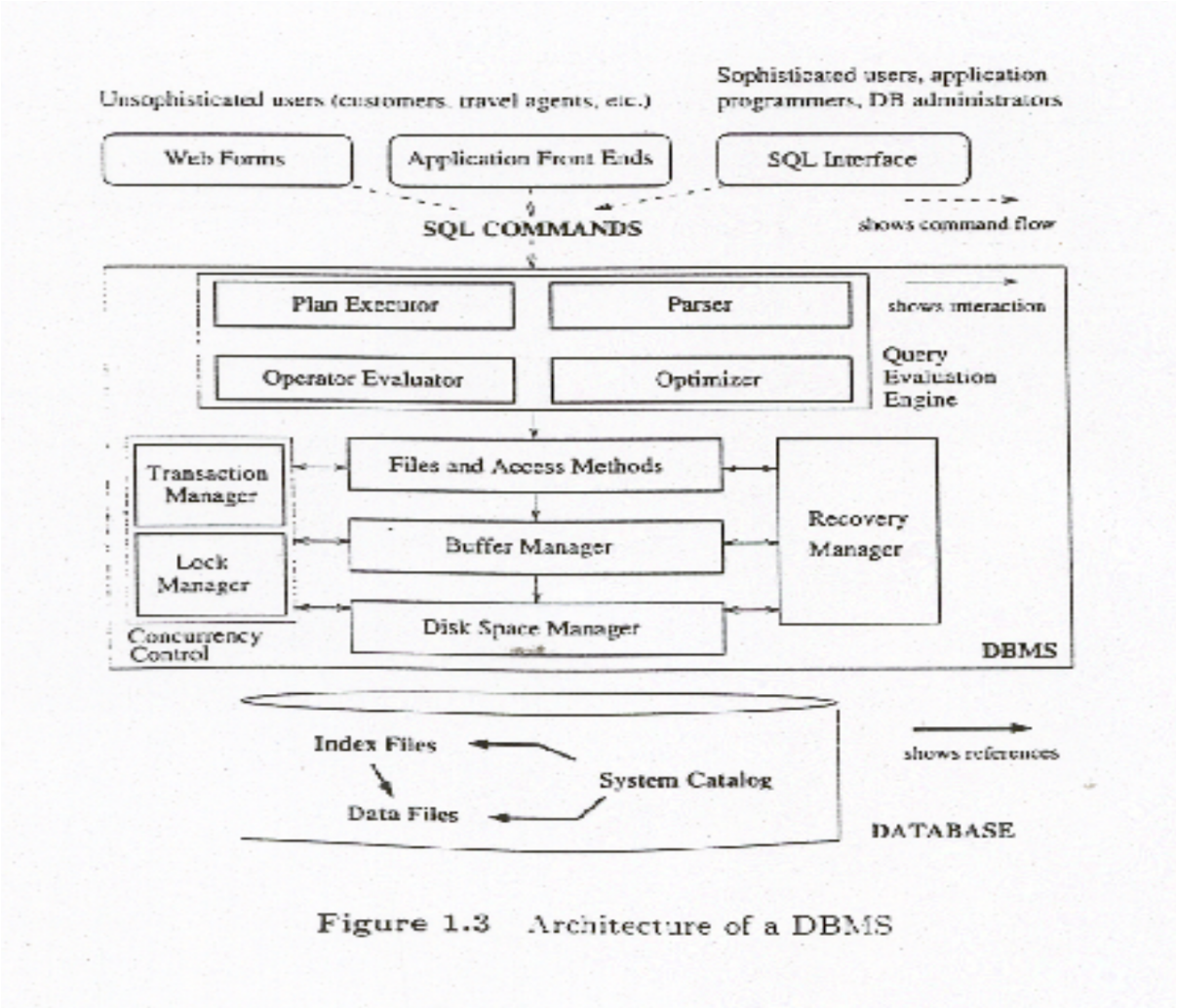


Figure 1.3 Architecture of a DBMS

0. Intro

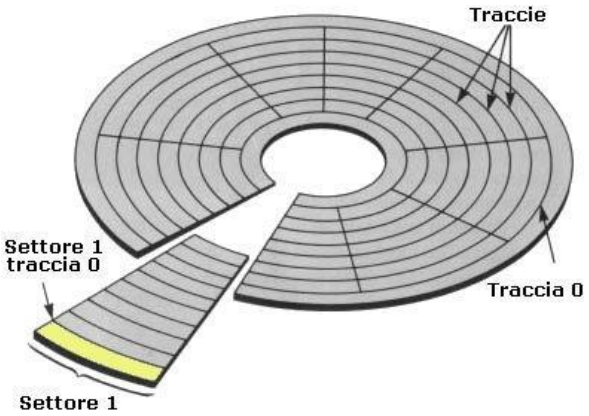
- **Database**: Collezione di dati organizzati per consentire dei Task di retrieval, analisi e visualizzazione dei risultati
- **DataModel**: formalismo per descrivere la struttura dei dati che popolano un DB (relazionali, oggetti, grafo..)
- **Schema**: particolare insieme di vincoli sui dati

1. Organizzazione dei file

I dati gestiti da un DBMS sono salvati su memoria solida (HDD).
Per eseguire operazioni di READ/WRITE su questi dati bisogna prima portarli in memoria principale (RAM) e poi successivamente di nuovo in memoria solida (in caso di write).
I dati non si salvano direttamente sulla memoria principale poiché questa è volatile e quindi a ogni riavvio andrebbero persi. C'è anche la questione del costo: a parità di quantità costa molto di più la memoria RAM.

Disco

I dati in memoria solida sono salvati in unità chiamate **disk blocks** o **pages**.



Per elaborare i dati sull'HDD c'è una testina che si sposta andando in avanti o indietro per raggiungere una certa traccia e su e giù per andare a contatto col disco. Un disk block corrisponde in un insieme di settori contigui su una traccia.
Per una maggiore efficienza i disk block di un file devono essere messi adiacenti così da minimizzare il delay della rotazione del disco sfruttando eventualmente anche il **prefetching**.
I tempi per accedere ad un disk block sono:

1. seek time (moving arms to position disk head on track)
2. rotational delay (waiting for block to rotate under head)
3. transfer time (actually moving data to/from disk surface)

Dati in memoria esterna

- Memorie a confronto:
    - HDD: può ottenere pagine in ordine random a un costo fisso anche se conviene sempre posizionare pagine in maniera consecutiva come detto prima
    - TAPES: possono solo leggere in sequenza e sono più economici degli HDD
- File organization: come vengono organizzati i record su file:
    - record id
    - indici
- Buffer Manager: fa da buffer tra la memoria principale e memoria solida

SYSTEM CATALOG: è un componente del DBMS che consiste in un insieme di tabelle e views che definiscono informazioni sul database stesso come ad esempio il tipo di indici usato, le varie relazioni, etc

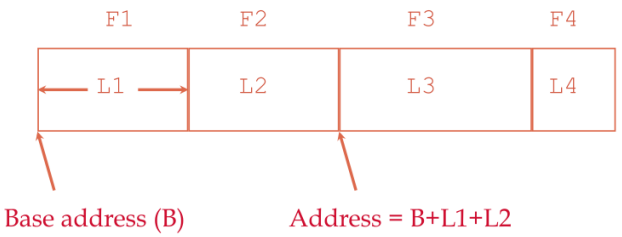
File Organization

Un file in ambito DBMS è una collezione di pagine, ogni pagina contiene i record, ogni record contiene più campi. Per file organization si intende il metodo con cui salvare i record dei file in memoria solida. Tra le più comuni ci sono:

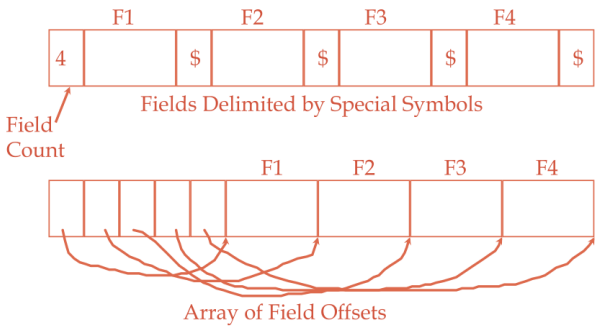
- RID: è il metodo più stupido, viene solamente salvato l'id del record che servirà per accederci fisicamente
- HEAP: strutturato quindi a pila ed è efficiente quando l'accesso comprende una lettura del file per ottenere tutti i record (lettura sequenziale infatti)
- SORTED FILE: ordina i record secondo un certo ordine specificato
- INDEXES: organizza i record tramite alberi o hashing

Formati dei campi di un record

- LUNGHEZZA FISSA: ogni record ha lunghezza fissa e così facendo per accedere ad un determinato campo del record l'operazione è semplice, basta aggiungere all'indirizzo di partenza del file la lunghezza di ogni campo. Le informazioni riguardo i tipi dei campi sono salvati nel SYSTEM CATALOG



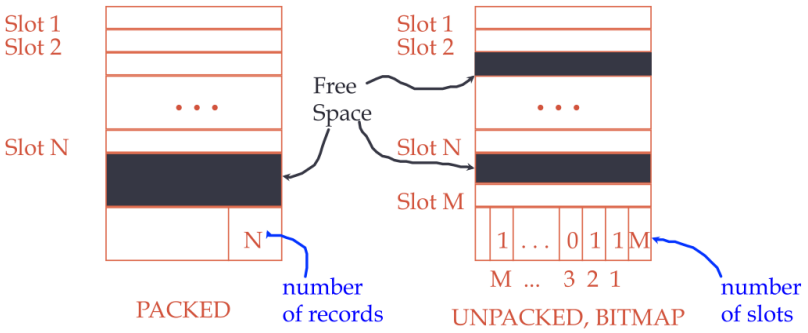
- LUNGHEZZA VARIABILE: qua ci possono essere due scelte: o ogni campo è separato dall'altro con un delimitatore speciale oppure tutti gli offset dei campi sono salvati a parte



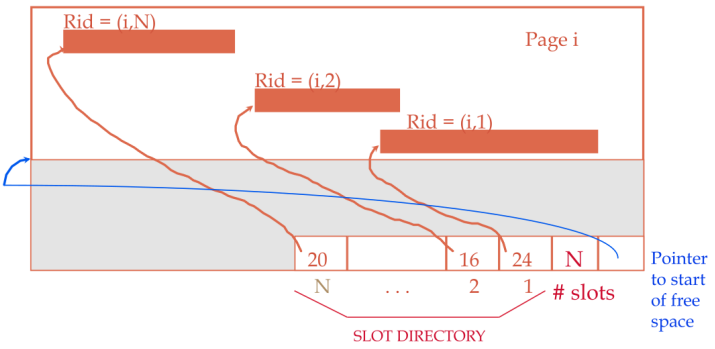
Formati dei record dentro le pagine:

Ogni record in una pagina è identificato da RID composto da <page id, slot number>. Una pagina è una collezione di slot, uno per ogni record

- RECORD A LUNGHEZZA FISSA:
  - possono essere posizionati uno dopo l'altro, lasciando spazio libero sempre al fondo: quindi bisogna spostare dei record se ad esempio uno a metà viene eliminato, obbligando quindi a cambiare anche RID (se ho un indice devo cambiare anche il puntatore degli indici, operazione costosa perché faccio operazioni di I/O)
  - oppure a caso, quindi permettendo dei buchi dentro il file, ma bisogna gestire le informazioni di quali slot sono pieni e quali sono vuoti → uso una directory.



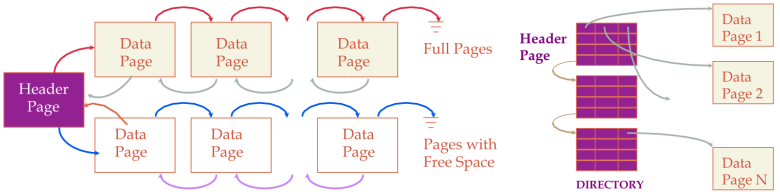
- RECORD A LUNGHEZZA VARIABILE:
  - ho una directory come il secondo metodo elencato sopra. in questa directory tengo i puntatori tra il record e l'inizio dello spazio in memoria in cui risiede. + dei puntatori per l'inizio degli spazi liberi (ne posso avere di più a seguito di cancellazione di spazi occupati). A fronte di una cancellazione posso: rendere lo spazio cancellato come disponibile senza fare altro. Oppure posso compattare tutto rendendo di fatto lo spazio libero sempre al fondo (come il primo metodo elencato sopra). Il compattamento non ha downside rispetto al caso sopra: non devo modificare gli indici, ma mi basta solo cambiare l'elemento della directory perché gli indici puntato alla directory. Il compattamento in se, come nel caso sopra, non e' costosa perche sono in memoria principale.



Heap File

Il funzionamento rimane uguale a quello di un heap ma ci sono diverse implementazioni:

- AS LIST: c'è una pagina header e tutte le altre pagine hanno un puntatore alla pagina successiva e a quella precedente
- PAGE DIRECTORY: ci sono più header page che contengono solo puntatori alle pagine di dati effettive. Questa soluzione risparmia memoria rispetto a quella prima vista



Indici

Servono per velocizzare la ricerca di un record dato un valore di un campo su cui si è costruito l'indice. Un indice contiene un insieme di data entries (e di index entries) in cui al loro interno possono essere contenuti diversi tipi di dato:

1. Record completo con la chiave *k*: In questo caso l'indice rappresenta in tutto e per tutto l'organizzazione dei dati dei record. Al più ci può essere solo un indice di questo tipo per una relazione. Comporta data entries di grande dimensione, e di conseguenza un numero grande di pagine per contenerle, di conseguenza comporta la gestione di informazione aggiuntiva dentro l'indice
2. <*k*, RID>: data entries piu piccole e quindi un numero di pagine minore. I record veri e propri sono gestiti con tecniche di organizzazione come heap o sorted file
3. <*k*, List of RIDs>: più efficace in termini di memoria che il caso 2 ma le data entries sono a dimensione dinamica (possono superare la dimensione della pagina: come gestisco? pagine di overflow o creo una seconda data entries con la stessa chiave)

ISAM

E’ una tipologia di indice in cui l’albero che viene creato e’ statico, ovvero i nodi interni (le index entries) non cambiano mai. I nodi foglia contengono le data entries e dato che l’albero e’ statico bisogna gestire gli overflow delle foglie.
Un file con questo indice ISAM viene creato inserendo prima le data pages, ovvero le pagine delle foglie, ordinate per la chiave dell’indice in maniera sequenziale. Poi vengono inserite le index pages contenenti le index entries e infine lo spazio allocato per le overflow pages. In questo indice non servono alle foglie i puntatori alla next foglia, in quanto le data entries sono allocate sequenzialmente su file quindi basta andare avanti con lo scan.

B+ Tree

Struttura ad albero come ISAM, ovvero index entries e data entries, ma in questo caso l’albero e’ dinamico e le foglie contengono un puntatore alla next foglia. L’albero essendo dinamico comporta delle operazioni di riorganizzazione in seguito ad insert o delete.

Prefix key compression

Tecnica che permette di comprimere le chiavi all’interno di una index entries, permette di aumentare il fan-out, ovvero il numero di puntatori che un nodo interno può avere.
L'ordine di un albero indica l’occupanza minima e massima di ogni nodo:
$d <= m <= 2d\ entries$ con d= ordine albero. In pratica il concetto di ordine non viene utilizzato, rimpiazzato dall’occupanza fisica di un nodo. Questo perché le index entries possono contenere molti più elementi che le data entries; le chiavi di ricerca differenti possono avere dimensione differente; ed infine se si utilizza il caso 3 non si sa a priori la dimensione di una data entries.

Bulk loading

Tecnica che permette di creare efficacemente un indice B+ su una certa chiave. Si parte ordinando tutte le data entries e iterando su questa lista creando un puntatore (ed eventualmente una index entries) per ogni data entries. Essendo che parto dopo aver ordinato, gli split avvengono sempre e solamente a destra dell’albero, facendomi risparmiare tempo rispetto a creare l’albero con inserimenti multipli: meno lock, piu concorrenza. In oltre bulk loading mi mantiene le foglie allocate sequenzialmente (isam like, inserimenti multipli no) ed essendo che posso decidere quando fare gli split, posso scegliere un fattore di riempimento iniziale.

Classificazione di indici

Primary vs Secondary

Se la chiave dell’indice e’ primaria o secondaria

Clustered vs Unclustered

Se i data records sono ordinati come (oppure l'ordine e' molto simile, questo nel caso delle overflow pages) le data entries, allora si parla di indice clustered. Il caso 1 implica un indice clustered. Nel caso unclustered le data entries sono ordinate mentre i data records sono sparsi all'interno di un heap file.

Gli indici possono anche essere composti da più chiavi.

Indici Hash

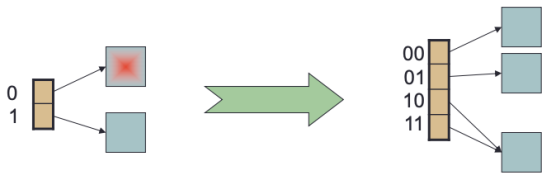
Sono una collezione di bucket dove ogni bucket al suo interno ha una primary page e nel caso anche delle pagine di overflow. Questi bucket gestiscono le data entries.
C'è una funzione di hashing che calcola in che bucket ogni data entry appartenga, il calcolo avviene sulla base della chiave di ricerca. Non esistono index entries in questo tipo di indice.

Static Hashing

Il numero di primary pages è fisso ed allocate sequenzialmente e mai deallocate. La funzione di hashing è la semplice h(k) mod M dove M è il numero di primary pages. Ci sono naturalmente anche pagine di overflow e il problema infatti e la loro dimensione poiché possono diventare molto grandi. Se e' troppo pieno rischio di dover rifare l'indice da 0 con una nuova funzione di hash.

Extendible Hashing

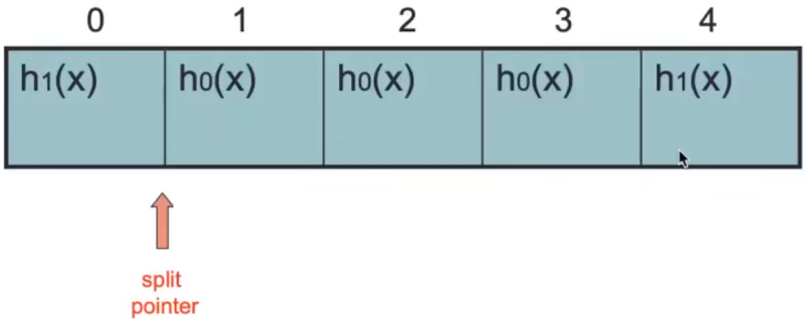
Cercano di risolvere il problema di overflow. Viene utilizzata una directory che raddoppia di dimensioni a richiesta. La directory contiene un'associazione al suo relativo bucket. Se il bucket di una directory e' pieno, vengono raddoppiate le directory e viene creato un nuovo bucket a cui le nuove directory puntano. Anche la funzione di hashing ovviamente cambia visto che ho un numero di directory e bucket diversi. C'e' il rischio che non avvenga una distribuzione equa dei dati dentro il bucket pieno, costringendomi a raddoppiare tante volta senza risolvere veramente il problema



Linear Hashing

Ho una serie di bucket ed uno split pointer. Quando raggiungo il limite di pienezza di un qualsiasi bucket porto avanti lo split pointer (quindi se lo split pointer e' al bucket 0, lo porto al bucket 1 (anche se il bucket pieno e' tipo il quarto)). Portare avanti lo split pointer vuol dire splittare il bucket a cui punta. Viene quindi creato un nuovo bucket (es.: se parto che ho 5 bucket, viene creato il bucket 6) e sia il bucket 1 che il bucket 6 avranno una nuova funzione di hash *h'* associata, mentre i bucket rimanenti hanno quella originale *h*. Quando poi un ad altro bucket serve lo split, muovo lo split pointer al bucket 2, e ora sia il bucket 2 che il nuovo

bucket 7 splittato dal 2 si vedranno usare la nuova funzione di hash di prima, ovvero *h'*. Procedo cosi fino a che tutti i bucket usano la funzione *h'*. Quando tutti la usano, il puntatore ritorna a 0 (e' un circolo) e se serve fare un altro split uso una nuova funzione *h"*. Procedo cosi in maniera circolare.
Il problema di questo approccio e' che se il bucket che ha bisogno di split si trova lontano dallo split pointer, questo mi costringe a fare tanti split inutili prima dello split che volevo fare.



(foto che non riflette il mio esempio) :D

Modello dei costi

B: The number of data pages in the data file
R: Number of records per page
D: (Average) time to read or write disk page

Assunzioni:

- Alt (2), (3): data entry size = 10% size of record (10% di 1.5 => 0.15)
- Hash: No overflow buckets.
    - 80% page occupancy => File size = 1.25 data size
- Tree: 67% occupancy (this is typical). (67% => ⅔ => 3/2 =>1.5)
    - Implies file size = 1.5 data size

	(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

Heap

- Scan: leggo tutte le B pagine quindi → B * D
- Equality: mediamente il record che voglio trovare si trova a meta' (caso ottimale si trova alla prima pagina, caso pessimistico si trova all'ultima pagina, in media a meta'): quindi devo scansionare meta' pagine → 0.5B * D
- Range: dato che non ho ordinamento devo scandire tutto → B * D
- Insert: costo di trovare la pagina libera + costo di inserimento. La pagina libera può essere al fondo dell'heap oppure se si usa una directory mi basta trovare uno slot che punta ad uno spazio vuoto → D + D → 2D
- Delete: costo di equality + cancellazione del record: *search* + D

Sorted

- Scan: leggo tutte le B pagine quindi → B * D
- Equality: posso usare un algoritmo efficiente come la ricerca binaria → Log_2B * D
- Range: simile a equality, ricerco il primo valore con equality del range e poi proseguo sequenzialmente in avanti fino alla fine del range → (Log_2B + #pgs) * D
- Insert: costo di ricerca di dove inserire + costo di traslare le pagine in avanti per mantenere l'ordine (di nuovo, mediamente l'inserimento avverrà a metà)
  → *search* + 2 * 0.5B * D. faccio 2 * 0.5B * D perché le meta' pagine da spostare prima le leggo e poi le riscrivo, quindi → *search* + BD
- Delete: stesso ragionamento dell'insert → *search* + BD

Clustered

- Scan: Per la scansione non uso l'indice partendo dalla radice. Ma dato che si parla di indice clustered, alle foglie non ho un puntatore al record nel file di dati, ma l'intero record. Quindi le foglie dell'albero rappresentano di fatto il mio file di dati, di conseguenza posso partire con la scan direttamente dalla prima foglia e andare avanti sequenzialmente. Dato che le pagine sono piene a ⅔ (67%), per poter leggere 2 pagine piene mi occorrono 3 caricamenti → maggiorazione per un fattore di 1.5. Quindi il costo totale e' 1.5B * D
- Equality: costo di ricerca in un albero. Il costo quindi dipende dall'altezza dell'albero → Log_F 1.5B * D (devo sempre considerare che le pagine sono piene al 67%. F rappresenta il fanout, fanout più alto porta a alberi più bassi. compressione di chiavi ecc)
- Range: stesso discorso di prima con sorted: costo di ricerca + costo di andare avanti in sequenza → (Log_F 1.5B + #pgs) * D
- Insert: costo di ricerca + costo di inserimento → *search* + D (non devo fare riorganizzazione dell'albero perché assumiamo che le pagine sono piene al 67%)
- Delete: costo di ricerca + costo di cancellazione → *search* + D (anche qui niente riorganizzazione perché le pagine sono piene al 67%)

Non Clustered

- Scan: Le foglie contengono i puntatori al file di dati. Ci sono due opzioni
  - se voglio scandire senza ordinamento, mi conviene fare la scansione come heap, quindi leggendo direttamente il file dati senza passare per l'indice → B * D

    - se voglio scandire con ordinamento, devo usare l'indice ma ogni entry nelle foglie mi porta ad avere un accesso alla pagina del file di dati: quindi costo di attraversare ogni pagina delle foglie + costo di prendere le pagine del file di dati. Dato che le foglie contengono solo chiave-puntatore e abbiamo assunto che le data entries occupino solo il 10% del record effettivo (che era 1.5B), abbiamo che il costo di attraversamento delle foglie e' B * D * 0.15. Per ogni record dentro le pagine delle foglie devo andare sul file di dati, quindi il costo e' R * B * D. In totale abbiamo B * D * 0.15 + R * B * D → B * D * (R + 0.15)
- Equality: costo di ricerca dentro l'albero (considerando che le data entries sono solo il 10% del record totale) + costo di accesso al file di dati → Log_F 0.15B * D + D → D * (1 + Log_F 0.15B)
- Range: stesso ragionamento, mi posiziono all'inizio delle foglie e poi faccio accessi al file di dati. D * (#pgs + Log_F 0.15B)
- Insert: costo di ricerca + inserimento in indice + inserimento in file di dati → *search* + D + D → *search* + 2 * D
- Delete: costo di ricerca + cancellazione indice + cancellazione file di dati → *search* + 2 * D

Hash

- Scan: come per unclustered ho due tipi di modi per accedere. Nel caso voglia usare l'indice (che non ha senso perché i bucket non sono ordinati), dato che i bucket sono pieni all'80%, per leggere 4 pagine piene faccio 5 accessi → incremento di un fattore di 1.25. Quindi costo totale e' dato dal costo di attraversamento dei bucket + costo di prendere la pagina per ogni record nel bucket → B * D * 0.125 + R * B * D → B * D * (R + 0.125)
- Equality: costo di accesso al bucket + costo di accesso al file di dati → D + D → 2D
- Range: dato che i bucket non sono ordinati, devo fare una scan completa, quindi ho uso l'approccio tipo heap a costo B*D oppure usando l'indice hash a costo B * D * (R + 0.125)
- Insert: costo di ricerca bucket + costo di inserimento nel bucket + costo di inserimento in file di dati → *search* + D + D → *search* + 2 * D
- Delete: costo di ricerca + cancellazione nel bucket + cancellazione nel file di dati → *search* + 2 * D

2. Gestione del Buffer

Tecniche di rimpiazzamento

Vengono usate per rimpiazzare blocchi del buffer con nuovi blocchi e questo permette di ridurre gli accessi su disco. Viene tenuto traccia di un dirty bit che indica se la pagina è stata modificata e un pin count che indica quante volte è stata richiesta la pagina dalle transazioni. Se questo counter è a 0 (si chiama unpinned) la pagina è candidata per il rimpiazzamento

MRU

Most recently used, rimuove la pagina più recente. In questo caso per gestire questo algoritmo basta un bit, setti a 1 la pagina che sta venendo usata e tutte le altre a 0 (oppure un registro che tenga conto dell'id della pagina usata per ultima). Non soffre di sequential

flooding poiché si andrà sempre e solo a rimpiazzare un pagina nel caso di # buffer frames < # pages in file visto che sarà sempre quella rimpiazzata la più recente usata

LRU

Least recently used, rimuove la pagina usata da più tempo. E' costoso mantenere questo algoritmo, servirebbe un timestamp per ogni pagina e cerchi il più piccolo. Soffre del problema del sequential flooding quando si usano le pagine in maniera circolare e si toglie una pagina che serve immediatamente allo step dopo.

Clock

Approssimazione euristica di LRU meno costosa. Soffre di sequential flooding.
Considero le pagine come disposte circolarmente. Per ogni pagina memorizza il *PinCount,* ovvero il numero di transazioni che usano la pagina (se =0 vuol dire che non e' utilizzata e quindi posso pensare di rimpiazzarla) ed il *Referenced*, ovvero il bit di seconda chance. Durante il giro circolare se una pagina ha pincount a 0 e referenced a 1, può avere una seconda chance e setto reference a 0. Al giro successivo se nessuna transazione sta usando la pagina, quindi pincount a 0, posso toglierla in quanto referenced sta a 0.

Gestione a livelli

In generale gestire tutte le pagine come se fossero uguali e' sbagliato, ad esempio la pagina della radice di un indice e' usata molto di frequente, non ha senso toglierla dal buffer come se fosse una pagina qualsiasi. Vengono quindi adottate misure per gestire le pagine su differenti livelli.

Domain Separation (Abstract)

Alloco buffer diversi per task diversi; se un buffer non ha spazio, rubo da un altro.
Problemi: non cattura dinamicità delle query (alcune query sono più frequenti di altre) e non da importanza diversa a tipi diversi di pagine (es: considera in ugual maniera la radice di un indice rispetto a delle data-pages). usa la stessa politica ovunque (LRU).

Working Set (Intro)

La priorità di una pagina e' data dalla frequenza di utilizzo della sua relazione. Ogni relazione ha un suo working_set.

New Algorithm

Divido il buffer in pogol e alloco per relazione, chiamati resident_set. Quindi ogni relazione ha un buffer e non può essere completamente sostituito.
Questi set sono dentro una lista ordinata in base ad una priorità (che può essere data dalla frequenza di accesso di una relazione) dove al top della lista inserisco prima le pagine libere e poi i resident_set con priorità più bassa. Così quando devo sostituire parto dall'alto della lista e scendo fino a che non trovo una pagina valida ad essere rubata. Quando rubo la porto dentro il mio resident_set. MRU viene usato per ogni relazione (non molto bello).

Hot Set

Assegnare i buffer non alle relazioni ma alle query. Per assegnare la dimensione ci si basa sugli hotset, ovvero il minimo numero di pagine del buffer che permette di minimizzare drasticamente gli accessi al disco per l'esecuzione della query. Usa LRU nel buffer della query, ed e' pessimistico (caused by LRU).

Query Locality Set Model

Simile a Hot Set, ovvero lavora in base alle query. Per ogni tipologia di operazione che fa la query identifica la migliore gestione del buffer

- straight sequential (ss) one page frame
- clustered sequential (cs) size of cluster
- looping sequential (ls) MRU
- independent random (ir) ss
- clustered random (cr) cs
- straight hierarchical (sh) one page frame
- sh+ss ss
- sh+cs cs
- looping hierarchical (lh) prob(p in ith level) = $1/f^i$

- Straight sequential(SS) - only need 1 page frame because once you 've used a page won't go back to it.
    - operazioni di scan o selezione senza uso di indici
- Clustered Sequential(CS) - size of the cluster needed, have sequential access but using more than 1 page at a time. So need all the pages accessed simultaneously.
    - approccio SS a cluster
- Looping Sequential (LS) - uso MRU → come SS ma quando arrivo alla fine delle pagine ricomincio dall'inizio. più frame disponibili nel buffer meglio e'
    - usato tipo nella relazione interna di un join table oriented
- Independent Random (IR) - Random access, no pattern, behavior is similar to straight sequential because once accessed it most likely won't be used again soon. Use LRU. 1 page frame
    - usato negli indici non clustered: attraverso le foglie e poi vado dove punta il puntatore
- Clustered Random (CR) - similar to cluster sequential but have to estimate the size of the cluster. size of the cluster needed
    - approccio IR ma a cluster
- Straight Hierarchical (SH) - Start at the top of index structure and go down because at every level only use 1 page
    - navigazione di indice, quindi nelle operazioni dove viene usato un indice
- SH followed by SS - B+ tree is sorted and sequential, so go down tree then follow across the pages so similar to SS. 1 frame is enough
    - query di range con indice, arrivo alla foglia e vado avanti in sequenza attraverso le foglie. usato nelle index only query (ovvero le query dove mi basta l'uso dell'indice)
- SH,CS - basically clustered pages. size of cluster needed
    - stessa idea di prima ma a cluster

- Looping Hierarchical - start from the top, go to a data element, go back to top look for next one, go back to top, etc. So upper level pages are accessed more often:
    - Prob(p in i^{th} level) = $1/f^i$
    - 2nd level accessed 2x as much as 1st level, 3rd level 2x as much as 2nd, etc.
        - quando ciclo più volte sullo stesso indice (loop interno di un join con uso di indice). più pagine ho meglio è. perché visto che loopo più volte e' meglio tenersi le pagine in memoria. La radice verrà' sempre usata quindi non ha senso toglierla. più scendo dall'albero, più bassa e' la probabilità di usare la pagina di quel nodo.

DB Min

Si basa sulle file instance, ovvero per ogni file di dati ho una file instance (file_id, query_id). Al file instance ho anche associato una certa query che sta usando quel file di dati. In memoria invece ho le pagine del buffer e le pagine associate ad un determinato file instance appartengono allo stesso locality set (quindi le pagine usate dalla query). Se nel buffer una pagina non appartiene a nessun locality set vuol dire che e' libera (non più usata). Nel buffer una pagina ha al più un locality set. Ho anche una global table per condividere le pagine tra query. Ogni locality set viene gestito in maniera indipendente.
Quando una query richiede una pagina, ho 3 casi:

1. pagina nella global table (quindi in memoria principale) e nel mio locality set: e' mia e la uso.
2. pagina nella global table e in un altro locality set: cambio owner della pagina e mi metto io come owner. (ora viene gestita secondo la mia politica di rimpiazzamento).
    a. se in nessun locality set: e' libera (la query che la usava ha finito), la metto dentro il mio locality
3. pagina non nella global table: deve essere portata in memoria da disco e procedi caso 2

Quando devo rimpiazzare le pagine uso la mia tecnica di rimpiazzamento in base al pattern di accesso come in QLSM all'interno del mio locality_set.
Vantaggio: **posso condividere pagine tra query diverse**.

3. Valutazione delle query

Cercare il miglior query plan per l'esecuzione di una query. Per far questo viene memorizzato il System Catalog, che contiene metadati sulle relazioni (# record, # distinct valori, # pages ecc). Questo catalog viene aggiornato periodicamente.
In pratica non cerco il miglior piano ma evito i piani peggiori: troppo costoso fare una ricerca completa ed in più non ho tutti i dati sufficienti per fare calcoli precisi (system catalog e' limitato)

Access Path: metodo con cui vengono prese le tuple (file scan or index match (ovvero scan di un indice con condizioni di where)).
Un tree index match posso farlo sul prefisso delle chiavi dell'indice, sia con condizioni di uguaglianza che disuguaglianza. Un hash index match posso solo farlo con condizioni di uguaglianza sulla totalità della/delle chiavi. Cioè se ho un indice <a,b,c>, posso fare un tree index match su: <a,b,c> o <a,b> o <a>. non posso farlo su qualsiasi altra permutazione tipo

<c> o <b,c> ecc. Questo e' valido sia per condizioni di uguaglianza che disuguaglianza.. Ancora più vincolato se uso indici hash, posso farlo sempre e solo su <a,b,c> e solamente in uguaglianza.

Selezione

Le clausole delle condizioni vengono prima trasformate in CNF selezionando prima i record tramite indice e poi senza. Un metodo di ottimizzazione e' trovare l'access path più selettivo, che riduca al minimo le operazioni I/O. Il costo e' dato dal numero di pagine trasferite da disco a memoria centrale. C'e differenza fra indici clustered e unclustered. Nel primo caso se ho una selezione con una condizione di < oppure >, riesco a ottimizzare gli accessi perché le pagine di accesso sono consecutive. Nel caso di un indice non clustered, le data entries sono si consecutive, ma i dati reali possono essere su pagine completamente separate ed ogni volta carico una pagina diversa.

Proiezione

La parte costosa e' la rimozione dei duplicati se viene utilizzato la keyword DISTINCT.
Metodi:

- sorting approach: ordini le tuple sulla base dei valori della proiezione e poi rimuovi i duplicati. (o rimuovi direttamente mentre sorti).
- hashing approach: crei in memoria una struttura ad hash sulle chiavi della proiezione ed elimini i duplicati
- se esiste un index sulle chiavi di proiezione può essere più veloce fare l'ordinamento

Sort

Il sort viene usato frequentemente dal dbms: query di ordinamento, bulk loading di un b+ tree, eliminare duplicati, fare il sort-merge join. problema: come fare se la ram è poca e non può contenere tutte le tuple? Non si può usare virtual memory o swap perché sarebbe lentissimo.

2-way sort

Bastano 3 pagine di buffer: 2 per le due pagine di input, 1 per la pagina di output.
Ho una sequenza di pagine disordinate, dove ogni record dentro le pagine sono anche disordinate. Al passo 0 ordino i record dentro le singole pagine (es 5000 pagine). Al passo 1 accoppio pagine adiacenti, ordino i contenuti delle coppie mergiando, risultando in 2500 coppie di pagine ordinate. Al passo 2 creo quadruple di pagine, ordinando e mergiando (1250 quadruple di pagine). Continuo cosi fino alla fine. Il numero dei passaggi del merge-sort e' $\log_2 N + 1$ dove N è il numero di pagine totali. Il +1 viene dal passo 0. Per ogni passaggio leggo e riscrivo ogni pagina, quindi 2N(passo 0) + 2N * $\text{Log}_2 N$ (passo 1 in poi).
Costo totale: $2N * (\log_2 N + 1)$.

General External Merge Sort

Generalizzazione del two way sort. Dove ho a disposizione *B* pagine del buffer, B-1 per gli input e 1 per l'output. Più buffer ho, più sveltisco l'algoritmo: questo perché riesco a eseguire il sort-merge con run piu lunghi in una sola passata.
Al passo 0 genero N/B runs di B pagine. Dal passo 1 in poi mergio B-1 runs alla volta
Esempio, 1000 pagine da ordinare:

- two way merge:
    - passo 0: leggo e riscrivo le 1000 pagine ordinandole internamente
    - passo 1: 500 runs da 2 pagine
    - passo 2: 250 runs da 4 pagine
    - ecc
- 5 frame buffer:
    - passo 0: 1000/B = 200 → 200 runs di 5 pagine l'uno. il costo e' sempre la lettura + scrittura di 1000 pagine ma a differenza di prima genero già dei run.
    - passo 1: 200/B-1 = 50 → 50 runs di 20 pagine l'uno. dato che mergio insieme 4 (ovvero B-1) runs di 5 pagine l'uno
    - passo 2: 50/B-1 = 13 → 12 runs di 80 pagine + ultimo run di 40 pagine
    - ecc

Il costo totale è $2N * ((Log_{B-1} N/B) + 1)$. Il +1 arriva dal passo 0 dove leggo e scrivo tutte le pagine (quindi 2N). Poi mi rimangono N/B run quindi per ordinarli faccio $Log_{B-1} N/B$ operazioni (ogni volta leggo e scrivo quindi faccio * 2N)

Sort Interni

Per i sort interni posso usare o quicksort o heapsort.
Heapsort:

1. Inizio creando una coda con priorità (heap) con i primi B elementi
2. Muovo (quindi tolgo dall'heap) l'elemento più piccolo dentro l'output
3. Faccio per inserire un nuovo elemento *e* dentro l'heap:
    - se *e* è più grande degli elementi in output, lo inserisco ordinato nella heap
        - riprendi da 2
    - se *e* è più piccolo di qualche elemento in output, scarto *e* e sposto in output i restanti B-1 elementi del mio heap
        - fine

Questo algoritmo mi crea al peggio un ordinamento di B elementi. Al meglio un ordinamento con tutti ed N gli elementi. Caso medio 2B.

Si preferisce heapsort a quicksort perché: generalmente quicksort e' più veloce, ma il caso peggiore di quicksort e' O(n^2) mentre il caso peggiore di heapsort e' O(nlogn)
Run più lunghi = numeri di run minore = meno accessi al disco

B+ Trees for sorting

mi porto alla prima foglia dell'indice e prendo le data entries ordinate:

- caso unclustered: poco efficiente perché per ogni record su foglia carico una pagina da disco
- caso clustered: molto efficiente perché record nella stessa foglia sono anche nella stessa pagina su disco

Join

Fare un prodotto cartesiano tra R ed S e poi fare selezione con la condizione del join e' stra costoso. Cerco di ottimizzare inserendo la condizione del join durante lo scan delle tabelle.

Assumiamo le relazioni *R* ed *S*.
M = # pagine in R. *N* = # pagine in S. (NB: lei sulle slide ha scritto che sono tuple, ma sono pagine)
p_R = *# tuple per page di R.* p_S = *# tuple per page di S*

Simple Nested Loop Join

```
foreach tuple r in R do
    foreach tuple s in S do
        if r_i == s_j then add <r, s> to result
```

se pr e ps = 100. M = 1000 e N = 500.
Costo: M + p_R * M * N = 1000 + 100*1000*500 I/Os. = 50 MLN e 1000
Si può leggere come: per ogni pagina di R la carico in memoria (costo M). Poi per ogni record di R (itero p_R * M volte) carico TUTTE le pagine di S (costo p_R * M * N).
Conviene che la relazione interna sia più piccola così la itero poco. (se è piccola puo starci tutta in memoria, o comunque ho meno page fault)

Page-oriented Nested Loop Join

A differenza dell'altro, non carico le pagine di S per ogni record di R, ma carico le pagine di S solamente per ogni pagina di R.

Costo: M + M * N = 1000 + 1000*500 I/Os.
Quindi non ciclo per record di R ma per pagina di R. Tolgo p_R dalla formula.
Conviene che la relazione esterna sia più piccola.

Block Nested Loop Join

Non lavoro a livello di singola pagina ma a blocchi di pagine sulla relazione esterna. Quindi uso 1 pagina per la relazione interna S, 1 pagina per l'output e le restanti pagine per la relazione esterna R (B - 2 pagine quindi). Questo ottimizza perché ciclo sulla relazione interna un numero minore di volte.
Costo: M + (M/(B-2) * N)

Oppure posso allocare B-2 frame per S relazione interna e solo 1 per R relazione esterna.
Oppure posso allocarne meta' meta'.
Vantaggio di allocarne tante per R mi permette di ciclare meno volte sulla relazione interna S
Vantaggio di allocarne tante per S riesco ad utilizzare con una politica MRU pagine che erano già in memoria, quindi per ogni iterazione non devo andare a prendere una pagina da disco (se la relazione e' piccola riesco a farla stare tutta nei b-2 blocchi)

Index Nested Loop Join

Lo posso usare se ho un indice su S.
Cost: M + p_R* M * (costo nel trovare il match nell'indice di S)
Il costo di trovare match e':

- indici hash: circa 1.2 per trovare il bucket + 1 per record
- indici albero: da 2 a 4 per la discesa sulla foglia + 1 negli indici clustered oppure +1 per ogni match con S negli indici unclustered

L'uso di indici non e' favorevole nel caso di join totali, senza condizioni. Anzi peggiorano le performance se vengono usati rispetto agli approcci sopra descritti. Sono utilizzati nel caso in cui sia necessario fare join con condizioni in cui serve l'ordinamento.

Sort Merge Join

E' simmetrico, cioè non cambia se considero prima R o prima S.
Considero ora se ho condizioni selettive, per ottimizzare le operazioni di I/O posso sfruttare l'ordinamento.

Sorto le relazioni R ed S sulla base della colonna di join, mergio e printo in output.
Nel merging faccio una scansione sequenziale sia di R che di S, portando avanti il puntatore sulla base del match di join:

1. ciclo su R finché tupla di R >= tupla di S
2. ciclo su S finché tupla di S >= tupla di R

- Se i valori di una relazione sono tutti distinti (join su chiave primaria) faccio la scansione una volta sola per ogni relazione perché il puntatore andrà sempre avanti → caso simmetrico
- Se invece ci sono duplicati dentro le relazioni, il puntatore sta fermo su R (relazione esterna) mentre va avanti su S (relazione interna) perché devo coprirli tutti per il match con il record di R. Finito il ciclo su quel gruppo di record di S posso portare avanti il puntatore di R. Nel caso il prossimo record di R e' anch'esso un duplicato di quello appena scansionato, devo portare indietro il puntatore di S per scansionare di nuovo quel gruppo per il match. R viene scansionata una volta sola, la relazione interna S viene scansionata più volte: ogni gruppo di S viene scansionato una volta per match con un record di R. Caso non simmetrico.

esempio:
R S

1 1
1 1
2 1
2 2
3
se ho una situa del genere, scansiono il primo record di R, 1. Poi faccio 3 scan di S. Passo poi al secondo record di R, sempre 1. Porto indietro il puntatore su S e rifaccio le 3 scan. Porto avanti il puntatore di R, ho 2 ora. Adesso posso andare avanti anche su S.
Invece nel caso non esistano duplicati i puntatori andranno sempre e solo avanti.

Cost caso migliore: M log M + N log N + (M+N) : costo di ordinare le due relazioni + costo di match → ho un solo match per ogni record di R.
Cost caso peggiore: M log M + N log N + (M*N) dove per ogni record di R ho un match con S.

Refinement Sort Merge Join

Inizio il merge durante la fase di sorting. Faccio fino al passo n-esimo del sort-merge, sia per S che per R. Il numero di passi dipende da quanti frame ho nel buffer: se ho basta frame mi basta il passo-0. Il mio obiettivo e' arrivare ad una situazione in cui mi trovo al più con B/2 run per ogni relazione. Questo perché per ogni run io inserisco una sola pagina di quel run dentro il buffer. Quindi dato che ho due relazioni da considerare, ed ho B frame liberi, al max posso avere B/2 run per relazione. Nel caso non abbia abbastanza frame, devo quindi fare più passi dell'algoritmo di sort-merge ed arrivare ad un punto in cui ho ridotto a sufficienza il numero di run. I run poi al loro interno hanno *X* pagine. Le pagine ed i record all'interno delle pagine sono ordinati, quindi i run sono ordinati.
Ridotto il numero di run, a questo punto inizio a fare il join: porto in memoria la prima pagina di ogni run. Confronto i record delle pagine della prima relazione con i record delle pagine della seconda relazione e controllo se ci sono match, sfruttando il fatto che le pagine sono ordinate al loro interno quindi non devo scandire tutti i record di una pagina. Se un record non ha match sono sicuro che non lo avrà' mai, neanche su altre pagine di run futuri, dato che le pagine dei run sono ordinati. (Quindi quelli che non hanno un match vengono scartati → questo vuol dire che le pagine vengono lette una volta sola)
Durante la fase di match porto avanti i puntatori come il sort-merge join classico. Quando una pagina di un run e' finita, carico la prossima di quel run.

costo: leggere e scrivere ogni relazione al passo 0 generando *n* run costa 2(N + M) + piu costo per il merging ovvero (N + M). totale 3(N + M). Il costo e' lineare (caso migliore dove ho abbastanza frame di buffer e' non ho chiavi duplicate)

Hash Join

Non può essere usato se non con join di uguaglianza (ovviamente). ~~Simile a nested loop join~~. Carico B-2 frame di buffer sulla relazione R e uno per la relazione S (o viceversa).
Ho un match quando la funziona di hash mi ritorna lo stesso valore sia per i record di S che di R.

A priori creo un partizionamento per ogni relazione, quindi utilizzando una funzione di hash vado a creare i bucket per R e per S. Anche nel caso io avessi già un indice hash, potrei aver salvato nell'indice non il record completo, ma magari un puntatore o solo alcuni campi. A me serve avere nei bucket i record completi, e non solo il corrispettivo chiave-valore, per questo a priori si crea questo partizionamento (nel caso avessi un indice che contiene il record completo allora solo in questo caso non devo fare il partizionamento a priori).
Ovviamente il partizionamento si fa sugli attributi del join e di conseguenza i match di R e S si troveranno sullo stesso bucket, perché la funzione di hash e' la stessa. Quindi durante il loop di scan per il match sfrutto questa caratteristica.
Come per nested loop join porto in memoria al max B-2 pagine della relazione R, in particolare porto le pagine di uno specifico bucket, ma poi non faccio una scansione completa di tutte le pagine di S, ma carico in memoria solo quelle che la funziona di hash mi dice di portare. Dato che la funzione di hash mi ritorna lo stesso bucket sia per R che per S quando ho un match. Quindi se i blocchi di R hanno bucket 1, devo ciclare sulle pagine di S solo quelli che sono dentro il bucket 1.
costo: leggere e scrivere la tabella per entrambe le relazioni costa 2(N + M) + fase di match dal costo di (N + M). Costo totale = 3(N+M).

Un problema di questo approccio e' che le partizioni devono essere formante in maniera uniforme, altrimenti si rischia di avere partizioni troppo grosse che non stanno dentro i B-2 frame del buffer. Rischio di dover applicare hashing in maniera ricorsiva (quindi entra in gioco h2, dove partiziono una partizione per fare in modo che stia tutta in memoria)
Quindi e' consigliato

- numero partizioni <= B-1
- numero pagine della partizione più grande <= B-2

B buffer > sqrt(M). hash join e' più parallelizzabile di sort merge. ma non ritorna la relazione ordinata come sort merge

Recap

Nei join di equalita posso usare tutte le tecniche sopra. Nel index nested loop creo l'indice sul valore di equalita, stesso discorso su hash join. Nel sort merge join ho l'equalita come condizione di scan delle pagine dei run.
Nei join di non equalita (es <) non possono usare hash join o sort-merge join. block nested loop e' il migliore, mentre index nested loop rischia di avere troppe operazioni di i/o (nella versione non clustered)
Il vantaggio di una soluzione simmetrica e' che evito di fare un passaggio: non devo andare a scegliere l'opzione migliore.

4. Ottimizzazione delle query

Il DBMS deve, dato uno statement dichiarativo:

1. convertire statement dichiarativi in statement esecutivi
2. stimare i costi di ciascun statement esecutivo
3. scegliere quello a costo minore

Il vantaggio di usare sql e' che ha un numero limitato di operatori, di conseguenza il tipo di accessi in memoria e' anche limitato → quindi posso ottimizzare al meglio questi access path.

SELECT $t_1.a$, $t_2.b$
(DISTINCT,SUM(),COUNT())
FROM TABLE1 t_1, TABLE2 t_2
WHERE $t_1.a = t_2.b$ AND $t_1.c > 10$
GROUP BY $t_1.a$

One Heuristic:

- Apply Π, б first
- Apply Joins Next
- Apply Group By Last

$\Pi_{t_1.a,\ t_2.b}\ [б_{t1.c>10}\ (\text{TABLE1} \bowtie_{t_1.a = t_2.b} \text{TABLE2})]$

$\Pi_{t_1.a,\ t_2.b}\ [б_{t1.a = t2.b,\ t1.c > 10}\ (\text{TABLE1 X TABLE2})]$

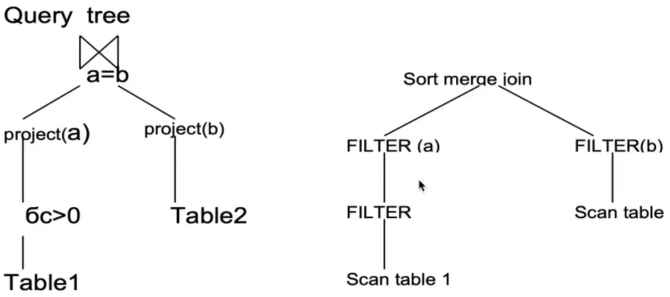
$\Pi_{t_1.a,\ t_2.b}\ [б_{t1.c>10}\ (\text{TABLE1} \bowtie_{t_1.a = t_2.b} \text{TABLE2})]$

$\Pi_{t_1.a}\ [б_{t1.c>10}\ (\text{TABLE1}) \bowtie_{a=b} \Pi_{t_2.b}\ (\text{TABLE2})]$

Il DBMS, dato uno statement, deve trovare dei piani fisici per operare la query → trovare il piano ottimo in uno spazio di ricerca (ovvero l'insieme dei piani possibili per quella query). Il query optimizer può lavorare in due modi:

- heuristic: fai prima le operazioni meno costose, questo permette di eliminare degli input per le operazioni più costose → problema: stiamo assumendo che possiamo cambiare l'ordine delle operazione in qualsiasi maniera liberamente (non sempre vero)
- cost based: ci basiamo su proprietà algebriche come la commutativa/distributiva. utilizza il cost model per stimare il costo della query. Il costo è definito come operazioni di I/O. problemi: come creare le alternative? qual'è un buon cost model? che statistiche usiamo?

Per la conversione di query in piani esecutivi creo prima degli alberi (logici), poi per ogni albero ho diversi alberi fisici, in quanto posso usare implementazioni diverse per la stessa operazione logica(es hash join o merge sort join)



a sinistra query tree logico, a destra query tree fisico.

Per i calcoli dei costi mi baso sui metadati delle relazioni (dentro system catalog). Ma devo anche tenere conta sulla forma che ha l'albero: se profondo a sinistra, bilanciato o profondo a destra. Un indice posso usarlo solo se ho un albero profondo a sinistra mentre un albero profondo a destra e' blocking. un albero bilanciato va bene se ho tanti core.

I metadati sono limitati, non posso pensare di tenere le statistiche di ogni combinazione di join tra tutte le tabelle, sarebbe troppo costoso il mantenimento. Mi tengo solo le info basilari e stimo poi la cardinalità dei join ad esempio.
Il costo globale di un piano logico o fisico è dato dal costo delle singole operazioni base (es: scan + filter + scan + filter + join).

Stima dei costi

I costi si calcolano tenendo conto di lettura tabella da disco (SCAN), tempo per la selezione (FILTER) e tempo per il join delle tabelle (JOIN)

B(R) = numero di blocchi
T(R) = numero di tuple
V(R.a) = numero valori distinti per a
V(R.a1----R.an) = numero valori distinti nel range a1---an considerati insieme

Scan

- Scan Cost
  - B(R) (unsorted)
  - B(R) + #of index pages (sorted on a clustered index)
  - T(R) + # of index pages (sorted on a non-clustered index)

chiaramente in uno scan l'uso di indici e' inutile

Selection

- Selection
  - S = $б_{a=c}$ (R) when "c" is a constant
  - T(S) = $\frac{T(R)}{V(R.a)}$
  - S = $б_{a<>c}$ (R) when "c" is a constant
  - T(S) = $\frac{T(R)}{V(R.a)}$ [V(R.a)-1] or T(S) = T(R) ????

T(S) = T(R) quando la condizione di disuguaglianza non matcha con nessun record.

- S = $б_{c1 \text{ or } c2}$ (R)
  - Assume c_1 and c_2 are independent
  - T(S) = $T(R) * [1 - (1 - \frac{n}{T(R)})* (1 - \frac{m}{T(R)})]$,
    - where "n" is the number of tuples satisfying c_1, and "m" is the number of tuples satisfying c_2

Io vedo al contrario, non prendo le tuple che non soddisfano entrambi

Join

- Join
  - R(x,y) ⋈ S(y,z)
    - T(R ⋈ S) = 0
    - T(R ⋈ S) = T(R)
    - T(R ⋈ S) = T(R)T(S)
  - $T(R \bowtie S) = \frac{T(R)T(S)}{\max(V(R,y),V(S,y))}$
  - $R(x,y_1,y_2) \bowtie S(y_1,y_2,z)$
  - $T(R \bowtie S) = \frac{T(R)T(S)}{\max(V(R,y_1)V(S,y_1))\max(V(R,y_2),V(S,y_2))}$

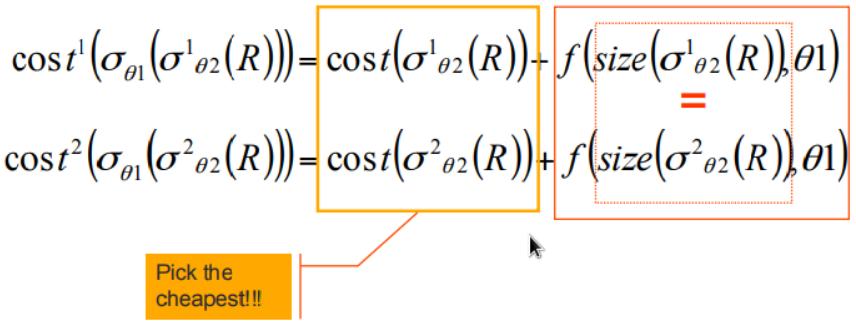
da 0 quando la condizione non ha match
da T(R) quando ho un left join
da T(R)T(S) quando ho un prodotto cartesiano

Assunzioni

Da un punto di vista del risultato, queste selezioni sono uguali. Da un punto di vista dei costi, hanno costi diversi

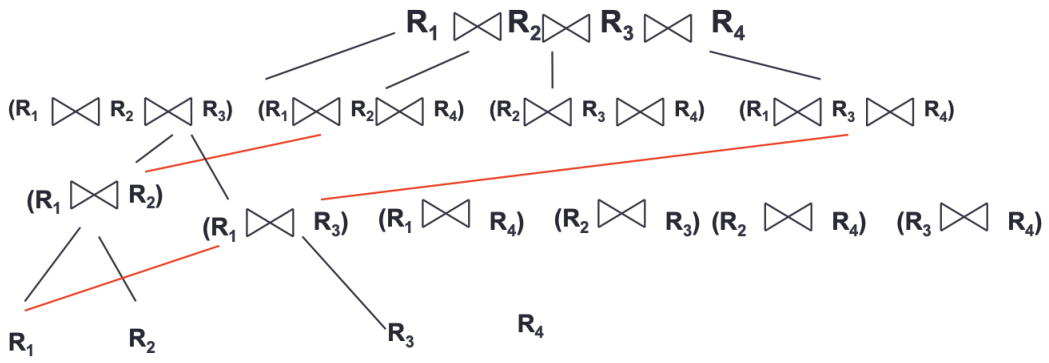
$$\sigma_{\theta 1 \wedge \theta 2}(R) = \sigma_{\theta 1}(\sigma_{\theta 2}(R)) = \sigma_{\theta 2}(\sigma_{\theta 1}(R))$$

Principio di sotto-query ottimalità: il piano ottimale di una query e' ottenuto trovando piani ottimali per tutte le sottoquery (divide and conquer). (Anche se non e' sempre vero, perche non ho mai tutte le informazioni nella stima dei costi, ma va comunque bene così)



Ricorsione

Dato il principio di sub-query ottimalità, ovvero l'ottimalità della query e' data dall'ottimalità delle sotto query, possiamo usare la ricorsione → dynamic programming.



Non calcolo da 0 ogni possibile combinazione, ma partendo dal basso inizio a calcolare i costi di R1, R2, R3, R4 per poi salire nell'albero andando a calcolare i join sfruttando i costi calcolati negli step precedenti. (Es quando calcolo R1xR2xR3 posso usare il calcolo fatto nello step precedente nel calcolare R1xR2).
Quindi
La soluzione ricorsiva top-down non è efficiente perché calcolo più volte la stessa cosa.
La soluzione dynamic-programming bottom-up molto meglio.

5. Transaction Manager

La concorrenza e' importante perche le operazioni di disco sono lente e nel frattempo e' utile far girare la cpu su altre transazioni per non sprecare tempo. una transazione e' una sequenza di read e write. Ogni transazione deve lasciare il db consistente come lo aveva trovato. La concorrenza e' ottenuta intervallando le operazioni di read/write delle varie transazioni → questo può portare a crash o situazioni di incosistenza. Una transazione conclude con il commit o con l'aborto

ACID

- **Atomicity**: una transazione può essere pensata come un'operazione atomica: o si esegue tutta o per niente.
- **Consistency**: una transazione arriva e trova il db consistente, viene eseguita e lascia il db consistente
- **Isolation**: l'esecuzione di una transazione e' completamente isolata dalle altre transazioni
- **Durability**: se una transazione fa commit, i dati devono essere resi persistenti

Recovery Manager garantisce A e D.

Inconsistenze

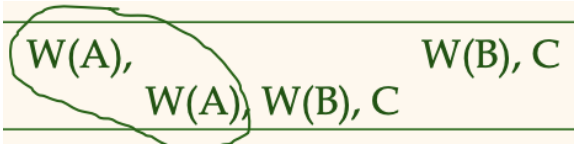
Letture sporche

T2 legge ma poi T1 abortisce

Letture non ripetibili

T1 ha due read di fila, devono dare lo stesso risultato, ma con T2 in mezzo non e' cosi

Perdita di aggiornamento



Quanto scritto da T1 viene sovrascritto da T2

Scheduling

1. Schedule Seriale: non interlaccia transazioni diverse
2. Schedule Equivalente: quando due schedule portano agli stessi cambiamenti di stato
3. Schedule Serializzabile: e' equivalente all'esecuzione seriale

Due schedule sono conflict equivalente se hanno lo stesso ordine di conflitti
Uno schedule è conflict serializzabile se ha lo stesso ordine di conflitti di uno schedule seriale.

Questo schedule non è conflict serializzabile, difatti il grafo delle dipendenze è ciclico

View Equivalent

Due schedule sono view-equivalente se:

- hanno lo stesso tipo di azioni sugli stessi oggetti
- If Ti reads initial value of A in S1, then Ti also reads initial value of A in S2

- If Ti reads value of A written by Tj in S1, then Ti also reads value of A written by Tj in S2
- If Ti writes final value of A in S1, then Ti also writes final value of A in S2

Se queste condizioni sono rispettate ed uno di questi due schedule e' uno schedule seriale mentre l'altro e' uno schedule con interfogliamento, so che quello interfogliato mi lascia il db in uno stato consistente. Il problema e' che dato uno schedule con *n* operazioni o *n!* possibili combinazioni, doverle confrontare tutte e' troppo costoso. Cerchiamo di restringere lo spazio di ricerca (approccio del grafo delle dipendenze e conflict serializable)

Grafo delle dipendenze

Grafo dove come nodi ho le transazioni e come archi i conflitti. I conflitti possono essere:

- read - write
- write - read
- write - write

Per ogni conflitto si traccia un arco che va da T1 a T2 se T1 precede T2. In Particolare:

- per ogni lettura ri (X) seguita da una scrittura wj (X), si traccia l'arco Ti à Tj
- per ogni scrittura wi(X) seguita da una lettura rj(X), si traccia l'arco Ti à Tj
- per ogni scrittura wi(X) seguita da una scrittura wj(X), si traccia l'arco Ti à Tj

Teorema: uno schedule è conflict serializzabile se il grafo è aciclico
La condizioni di aclicita e' sufficiente ma non necessaria: ovvero un grafo aciclico garantisce sempre che lo schedule sia conflict serializzabile (ovvero consistente), ma possono esistere schedule con grafici ciclici che sono comunque conflict serializzabili

Approcci pessimistici

Se il dbms si accorge che uno schedule non e' conflict serializzabile, dovrebbe terminare una transazione. Questo però' comporta spreco di risorse. Cerco allora di evitare i conflitti a priori usando il locking. Qualsiasi approccio di locking e' detto pessimistico. Svantaggi: overhead nel gestire i lock, possono verificarsi deadlock e contesa di lock.

Strict 2 PL

Strict two phase locking. Ogni transazione deve ottenere uno shared (S) lock se vuole leggere o un exclusive (X) lock se vuole scrivere. Piu transazioni possono ottenere S lock in contemporanea ma per avere X lock devi essere l'unico ad avere qualsiasi tipo di lock su quell'oggetto. I lock vengono rilasciati solamente a fine transazione (dopo abort o commit). Questo evita aborti in cascata. Strict 2PL permette solo grafi aciclici.

2 PL

Simile allo strict, ma una transazione non puo ottenere lock aggiuntivi una volta che ne fa cadere uno, di conseguenza i rilasci del lock possono avvenire anche prima che la transazione termini. Questo aumenta parallelismo ma puo causare aborti in cascata: se T1 rilascia lock su X, poi T2 assume lock su X e ci fa dei lavori, poi T1 abortisce → T2 deve anche abortire.

2 PL e Strinct 2 PL garantiscono che gli schedule siano sempre conflict serializzabili.

Lock Manager

Si occupa delle richieste di lock e unlock. Lock Table Entry tiene conto di: quali transazioni hanno il lock su quale oggetto, quale tipo di lock è ed un puntatore alle richieste di lock su quell'oggetto. Un lock shared può essere aggiornato in un lock exclusive.

Deadlock

Un ciclo di transazioni che aspettano il rilascio di lock a vicenda. Ci sono due modi per affrontare deadlock: prevention or detection

Deadlock prevention

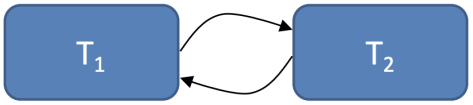
Assegna priorità alle transazioni come timestamp di arrivo. Assumiamo che T1 vuole assumere un lock che ha T2. Due metodologie:

1. wait and die: se T1 ha priorità più alta aspetta, altrimenti abortisce
2. wound wait: se T1 ha priorità più alta T2 abortisce, altrimenti T1 aspetta

Se una transazione ricomincia deve comunque mantenere il timestamp originale

Deadlock detection

Viene creato un wait-for graph dove i nodi sono le transazioni e gli archi sono: se T1 aspetta T2, ho un arco da T1 a T2. Periodicamente controllo il grafo e se ho un ciclo vuol dire che ho un deadlock. Nel caso di deadlock si abortisce una transazione → rischio di starvation

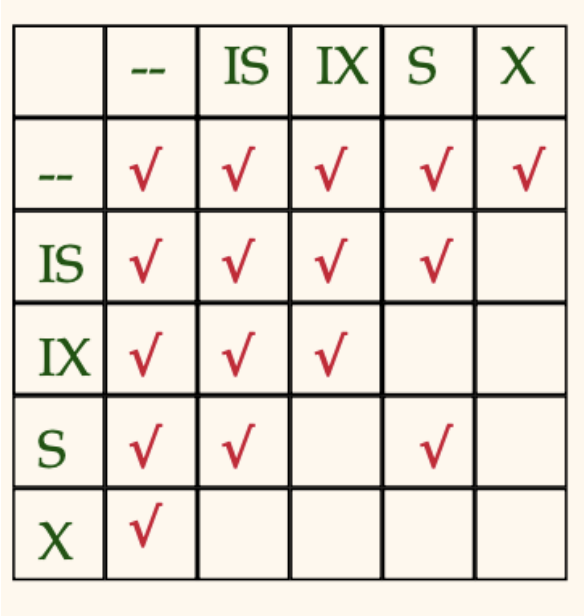


- T_1 è in attesa di un rilascio da parte di T_2 e T_2 è in attesa di un rilascio da parte di T_1.

Granularity Locks

Possibilità di fare lock a livelli differenti e introduzione di un nuovo tipo di lock.
Quando uso un Intention Lock su un oggetto, questo viene anche applicato a tutti i suoi ancestor. Es: voglio modificare un record. Metto IX su tutti gli ancestor del record (pagina, tabella, db ecc.) e metto X sul record stesso. Se nel frattempo arriva una transazione che vuole leggere tutte le row della tabella, prova ad acquisire lock mettendo S sulla tabella e IS su i suoi ancestor. Gli IS sugli ancestor può metterli dato che sono compatibili con IX, ma non può mettere S sulla tabella perché c'è un IX.

L’utilizzo degli intention permettono di migliorare le performance. Senza intention avrei operato in questo modo: la prima transazione mette lock X sul record che vuole modificare. La seconda transazione, che vuole leggere tutta la tabella, controlla sulla Lock Table ogni tupla e se il record a cui fa riferimento quella tupla ha un X lock smette di leggere perché va in attesa. E’ immediato vedere come le performance sono migliorate nel caso degli intention, non devo più controllare che qualsiasi record della tabella non abbia un X lock, ma mi basta controllare solo il tipo di lock sulla tabella stessa (che in questo caso sarebbe IX, non compatibile con S).



	--	IS	IX	S	X
--	√	√	√	√	√
IS	√	√	√	√	
IX	√	√	√		
S	√	√		√	
X	√				

Dynamic Database

Il database cambia continuamente, non e’ corretto assumere che durante lo scan di una tabella da parte di T1 nessuna transazione T2 non inserisca nessun nuovo record. Il conflict serializability garantisce serializzazione solo quando l’insieme di oggetti e’ fisso.
Assumiamo una tabella con i campi rating e età. T1 vuole trovare il record con età massima e che abbia rating = 1. Se nel mentre che T1 cerca la tupla con quei criteri, T2 inserisce un nuovo record con età maggiore di tutti gli altri, T1 non e’ piu corretto rispetto al nuovo stato del db. Problema chiamato: aggiornamento fantasma
Due meccanismi: index locking e predicate locking

Index Locking

Se c'è un index tree caso 2 sulla chiave rating (quindi con le data entries che contengono l’associazione chiave-puntatore), T1 può lockare la pagina in questione (in questo caso dall’esempio di prima si intende la pagina contenente i record con rating=1), in questo modo nessun nuovo record viene aggiunto (se non esiste attualmente nessun record con quella chiave, bisogna lockare la pagina dove quel record verrà’ inserito nel caso ci fosse). Se non esiste un indice adatto si deve lockare tutte le pagine della tabella in questione.

Predicate Locking

Una generalizzazione di index locking (o si può vedere index locking come un caso particolare del predicate locking dove sfrutto il fatto che io abbia un indice). In questo caso locko tutti i record che soddisfando un certo predicato (es rating = 1). Ha un overhead alto rispetto a index locking.

Locking in B+ Tree

Come si locka una foglia di un albero? Prima soluzione: lockare le pagine mentre si attraversa l'albero → performance terribili perché già solo bloccare la root non consente ad altri di accedervi. Bisogna cambiare approccio tenendo in considerazione che i nodi interni richiedono lock in X mode solamente quando ho bisogno di fare uno split.

Simple Algorithm

- search: parto dalla radice e scendo. durante la discesa locko il figlio di un nodo con S e unlocko quello appena attraversato
- insert/delete: parto dalla radice e scendo, locko con X man mano che scendo (così se ho da fare splitting o unione dei nodi interni ho il lock). Durante la discesa controllo anche se il nodo e' safe. Se e' safe, ovvero sono sicuro che non ci sarà necessità di split o unione, unlocko tutti i nodi ancestor, perché so che se anche avro' uno split piu in profondita' non arriverà fino a quei nodi.

Better Algorithm

- search: come prima
- insert/delete: metto i lock S come il protocollo search durante la discesa. arrivato alla foglia setto X. se il nodo non e' safe rilascio i lock e procedo da 0 usando la tecnica dell'algoritmo precedente

Questo algoritmo spera che solo il nodo foglia sia da modificare (nessuna necessità di split o unione). Nel caso in cui non e' safe io faccio lavoro inutile mettendo lock S sui nodi interni per poi disfarli e metterli a X. In pratica e' meglio del simple algorithm perché non sono così frequenti i nodi non safe.

Even Better Algorithm

- search: come prima
- insert/delete: uso il protocollo insert/delete del simple algorithm ma usando IX al posto di X. Una volta arrivato alla foglia e se devo fare splitting o union trasformo qualsiasi lock IX rimasto lungo il cammino in lock X partendo dalla radice.

Meglio IX che X perché mi permette di parallelizzare meglio. IX e' compatibile con IS e IX. Meglio del better perché non spreco mai lavoro

Approcci Ottimistici

Non uso approcci di locking, ma vado a correggere nel caso si creano situazioni di inconsistenza. Se i conflitti sono rari posso ottenere maggiore concorrenza facendo un controllo sulla consistenza giusto prima del commit.

Optimistic CC - Kung Robinson Model

Le transazioni hanno 3 fasi: Read, Validate, Write.

- Read: legge da database ed eventualmente fa modifiche su copie private
- Validate: controlla che le copie private siano consistenti con il db
- Write: rende persistenti le copie private

La fase di validation è la più importante.

Ogni transazione ha un id numerico, come il timestamp. Il timestamp viene assegnato alla fine della fase di read. Per ogni transazione tengo in memoria un ReadSet ed un WriteSet, ovvero set di oggetti letti/scritti.
L'idea alla base di questo approccio è che: tutte le transazioni precedenti alla tua che sono già alla fase di validazione o qualsiasi fase successiva, sono conflict serializabile. Se può esserci un conflitto questo è dato dal fatto che la causa e' la mia transazione. Per questo motivo si usa come timestamp l'inizio della fase di validazione.
Assunzione: io sono Tj.

1. Test1: per qualsiasi transazione Ti < Tj, se Ti completa prima che inizi Tj → OK
2. Test2: // Ti < Tj, se Ti completa prima che Tj inizi la fase di write
    a. se WriteSet(Ti) intersezione ReadSet(Tj) e' vuoto → OK
    b. Non ho conflitti perché le Tj legge dati diversi (no dirty reads, ma possibili overwrites, ma gli overwrites mi vanno bene perché Tj viene eseguita dopo)
3. Test 3: // Ti < Tj, se Ti completa la fase di read prima che Tj completi la sua fase di read
    a. se WriteSet(Ti) intersezione ReadSet(Tj) e' vuoto AND
    b. se WriteSet(Ti) intersezione WriteSet(Tj) e' vuoto → OK
    c. Non ho conflitti perché le transazioni operano sue due dati diversi(no dirty data, e neanche overwrites)

Validazione Seriale
Se adotto validazione seriale, ovvero che la validazione (e anche la fase write) avviene in sezione critica, ovvero che non posso avere due validazioni in contemporanea da parte di due transazioni, ma una sola validazione alla volta, il caso 3 sopra descritto non può più accadere e posso farlo cadere, perché so che le write avverranno in maniera seriale e quindi e' come se fosse serial-equivalent. Svantaggio: la serial validation e' supportata se il sistema operativo lo supporta e minor parallelismo. Vantaggio: maggior semplicita

Svantaggi generali: può capitare starvation se una transazione ritorna sempre non valido nella fase di validazione. Maggior overhead per gestione dei writeset o readset, se validation fallisce la transazione riparte rendendo inutile lavoro fatto fino ad ora.

Optimistic 2PL

Setto lock S durante la lettura, se modifico faccio copie locali, nel momento in cui voglio scrivere persistente ottengo il lock X. Poi rilascio i lock. E' un misto fra kung robinson ed i metodi pessimistici. Non ha validation e quindi no restart, perché la transazione si blocca se non ha il lock X. Rispetto a kung robinson riduco di molto l'overhead però' devo gestire il deadlock dato che faccio locking.

Timestamp CC

Ogni oggetto ha un suo read-timestamp (RTS) e write-timestamp(WTS). Ogni transazione ha anche un suo timestamp(TS) settato quando incomincia. **Idea**: se una transazione T1 fa certe azioni A1 e transazione T2 fa certe azioni A2 e le azioni sono in conflitto e TS(T1) < TS(T2), allora A1 devono accadere prima di A2, altrimenti restart.

- Lettura:
    - se TS(T) < WTS(O): violazione → abort T e restart con timestamp più alto
    - se TS(T) > WTS(O): OK → T può leggere O, setta RTS(O) = max(RTS(O), TS(T))
        - quindi posso leggere l'oggetto solo se io sono arrivato dopo l'ultima scrittura
- Scrittura:
    - se TS(T) < RTS(O): violazione → abort T e restart con timestamp più alto
    - se TS(T) < WTS(O): violazione → abort T e restart con timestamp più alto
        - thomas write rule: se un'altra transazione scrive prima che scriva io e non c'e' stata nessuna lettura in mezzo, posso scrivere perché tanto in ogni caso la mia write va a sovrascrivere quella precedente (non è conflict serializable ma è comunque corretto)
    - else OK → T può scrivere O, setta WTS(O) = TS(T)
        - quindi posso scrivere un oggetto solo se nessuno lo ha letto da quando io sono arrivato e nessuno lo ha scritto

E' possibile garantire Isolation bufferizzando i write in attesa del commit della transazione, e se dei reader leggono l'oggetto non ancora committato aspettano. Altrimenti salvo diretto gli oggetti modificati prima del commit ma posso causare aborti in cascata se io abortisco e dei reader nel frattempo hanno letto.

Svantaggi: Overhead nei restart e nel mantenimento della persistenza dei timestamp degli oggetti in memoria fisica

Multiversion CC

Le transazioni si differenziano in transazioni che scrivono o transazioni che leggono. I writer creano delle copie di un oggetto e le salvano in un version pool mantenuto ordinato in base al timestamp. L'oggetto poi viene reso durevole durante il commit. Le letture sono sempre garantite, ma ci possono essere dei block in attesa che i writer committino, io infatti leggo dal version pool l'ultima versione, ma prima di procedere aspetto che ci sia il commit del writer.

Leggere:

Prendo la versione più recente di O tale che WTS(O) < TS(T). Se la transazione che ha modificato quella versione di O deve ancora committare, aspetto.

Scrivere:
Prendo la versione più recente di O tale che WTS(O) < TS(T).

- Se RTS(O) < TS(O) creo una copia aggiornata con le modifiche O' con WTS(O') = TS(T) e RTS(O') = TS(T). La scrittura e' bufferizzata nel version pool, in attesa del commit.
- Altrimento reject.

L'idea e che quando io arrivo lo stato e' consistente, quindi devo solo controllare che una mia scrittura non renda incosistente il db. Questo lo faccio controllando che nessuno abbia letto l'oggetto O dopo il mio arrivo.
Vantaggio: no intersezioni o set come kung robinson. gestisco solo i timestamp come timestamp cc
Se voglio mantenere isolation, i reader si bloccano se la versione che leggono non e' committata. Se non mi interessa isolation, i reader leggono subito un oggetto nel version pool anche se non committato → aumenta parallelismo ma c'e' rischio di aborto (se la transazione che ha creato la version abortisce → aborti in cascata)

6. Crash Recovery

Garantisce Atomicity e Durability → gestisce quindi i rollback e i casi di crash del sistema. Assumiamo che Strict 2PL e' attivo (evita aborti in cascata) e nessuna gestione di copie, si salva diretti su disco. L'idea generale e' disfare operazioni memorizzate su disco ma non committed e rifare operazioni non memorizzate su disco ma committed.

Gestione Buffer

La pagina di una relazione si trova su disco, ma può anche trovarsi su buffer. Se ci faccio una modifica, quando la porto su disco?

- Force: portata immediatamento su disco appena faccio commit. Vantaggi: semplice e comodo (no need del redo), ma sistema lento se ogni volta devo scrivere
- No Force: si porta su disco in un tempo indeterminato dopo commit. complica la durability: che succede se crasha tutto appena subito il commit ma non ho ancora portato su disco?

Politica di rimpiazzamento

- Steal: se buffer e' pieno ma a me transazione serve portare in memoria delle pagine, rubo un frame a qualcuno, quindi la pagina tolta viene salvata su disco. però se la transazione colpita non ha ancora terminato (violazione atomicity) e poi dopo abortisce e nel frattempo altri leggono la pagina rubata su disco? aborti in cascata (violazione isolation). Vantaggi: parallelismo ma complica la situa

- No Steal: se buffer e' pieno ma a me transazione serve portare in memoria delle pagine, aspetto. Vantaggi: semplice e comodo ma poco parallelismo (no need del undo)

Logging

Per gestire qualsiasi tipo di problema sopra descritto faccio logging a manetta. Il log viene scritto sequenzialmente quindi gli accessi sono ottimizzati anche se devo scrivere una grande quantità di dati. Ogni record di log contiene il minimo delle informazioni necessarie:
<XID, pageID, offset, length, old data, new data>
id transazione, pagina modificata, il punto della pagina modificata, lunghezza della modifica

Write Ahead Logging - WAL

Forzo la scrittura del log prima che la pagina modificata venga portata su disco (nel caso di uno steal) e forzo la scrittura di tutti i log subito dopo un commit. Garantisce atomicity e durability. Vuol dire che al momento del crash ho comunque il log aggiornato come le ultime modifiche(se sono committate). Quindi anche se uso una politica no-force dove non so quando porto le pagine su disco, se io faccio commit porto subito su disco almeno il log. Poi solamente dopo salvo la pagina modificata su disco (se c'e' crash posso fare i redo usando i log). Nel caso si usa politica steal dove posso portare in memoria pagine non committed, anche in questo caso scrivo prima il log e poi porto le pagine.

Ogni record di log ha un suo identificatore - **LSN**, Log Sequence Number (un numero auto incrementale). Ogni pagina contiene un **pageLSN**, ovvero l'ultimo LSN che ha toccato quella pagina. Il sistema tiene traccia in ram anche del **flushedLSN**, ovvero l'ultimo LSN flushato(portato su disco). Dato che uso WAL, nell'esatto momento in cui voglio portare su disco una pagina, il flushedLSN sara' >= del pageLSN di quella pagina(dato che la pagina viene portata su disco DOPO che io salvo il log su disco)

Log Record Filed

- prevLSN: punta al precedente LSN della transazione
- XID: id transazione
- type: update, commit, abort, end (quando il commit e' persistente su disco, oppure abort ha finito di fare undo), CLR (compensation log record, usati nella fase di undo)
- pageID
- offset
- length
- old data
- new data

Tabelle di supporto

Nella gestione dei log esistono anche le **Transaction Table** e la **Dirty Page Table** (tenute in ram)

La prima contiene un record per ogni transazione attiva: <XID, status, **lastLSN**>.
La seconda contiene un record per ogni dirty page nel buffer pool, ovvero una pagina modifica ma non su disco oppure una pagina rubata (in generale tiene traccia delle pagine non consistenti con il db), oppure una pagina modificata e su disco ma la transazione non ha ancora fatto commit. contiene <**recLSN**>, ovvero il primo LSN che ha modificato la pagina.

Assumiamo sempre che si usi Strict 2PL, le write su disco sono atomiche, WAL e steal/noforce

Checkpointing

Il DBMS fa periodicamente un checkpointing per minimizzare i tempi di recovery in caso di crash.

- begin_checkpoint record: indica quando e' iniziato il checkpoint
- end_checkpoint record: indica la fine del checkpoint.

Durante il checkpoint vengono rese persistenti le tabelle transaction table e dirty page table al tempo di begin_checkpoint. Anche tutte le pagine modificate da transazioni committed fino a quel momento sono rese persistenti.

Abort

No crash involved. Aborto controllato dall'utente. Devo disfare la transazione, ovvero disfare le cose fatte partendo dal punto di aborto fino al punto di inizio della transazione. Disfo solo quelle operazioni che sono state salvate su disco (se le pagine modificate sono solamente in ram non devo fare altro). Quindi:

- prendi lastLSN dalla transaction table
- disfo andando indietro usando il prevLSN di ogni record
- prima di fare la prima di operazione di UNDO, scrivo un record su log di tipo Abort

Per fare UNDO mi serve lock, ma uso Strict 2PL, quindi no problem, perché i lock vengono rilasciati da una transazione solo alla fine, ovvero dopo il commit o dopo la fine di un abort. Durante i vari UNDO scrivo un log di record di tipo CLR. Questo tipo di log contiene un campo aggiuntivo: **undonextLSN**, ovvero un puntatore al prossimo log di record da disfare (di per se questo undonextLSN = prevLSN del log di record da disfare). Questi CLR vengono usati nel caso di crash, cioe mi salvo le operazioni di UNDO che ho gia fatto, cosi se c'e' crash e devo ripetere l'aborto ho gia fatto alcuni UNDO e non devo rifarli. Alla fine di tutti gli UNDO scrivo END per indicare la fine completa dell'aborto.

Commit

Quando una transazione fa commit scrivo immediatamente su log un record di tipo commit. Dato che uso WAL, flusho tutti i record nuovi in memoria su disco. Questo garantisce che flushedLSN >= lastLSN della transazione. Poi trasferisco le pagine in memoria e dopo poi scrivo END sul log.

ARIES

In caso di crash, perdo il contenuto della ram, quindi anche la transaction table e dirty page table. Dato che faccio checkpoint riesco a recuperare quel contenuto. ARIES è un sistema che in 3 fasi fa ripartire il dbms ad uno stato consistente pre-crash. Le fasi sono

1. Analisi
2. Redo
3. Undo

Analisi

Ricostruisce lo stato delle tabelle di supporto fino al momento del crash partendo dal tempo di checkpoint. Parto dal record di end_checkpoint e vado in avanti: dato che il log rimane sempre aggiornato su disco, io prendo lo stato delle tabelle di supporto che ho salvato al tempo del checkpoint e lo aggiorno andando a scandire ogni record partendo dal checkpoint fino all'ultimo record con LSN più grande (l'ultimo prima del crash).

- se trovo un record di end: rimuovi la transazione dalla tabella delle transazioni (dato che c'e' end sono sicuro che le modifiche sono già persistenti)
- altri record: aggiungi XID alla tabella delle transazioni, set lastLSN = LSN
- record di update: se la pagina non e' in dirty page table la inserisco e setto recLSN = LSN (se la pagina e' già in dpt non devo aggiornare recLSN perché quel field si riferisce al primo LSN che ha modificato la pagina)

REDO

Adesso che ho finito l'analisi ho le tabelle di supporto aggiornate, quindi posso iniziare le fasi di redo/undo per portare lo stato del database e del sistema globale al momento del crash. Non considero più il checkpoint, perché quello mi serve solo per ripristinare lo stato della transaction table e dirty page table.
Per la fase di REDO parto dal record su log che ha come LSN il più piccolo recLSN della dirty page table, ovvero parto dal primo log che ha modificato per primo una pagina che e' finita di dpt.
Quindi scansiono andando avanti il log e' faccio REDO quando trovo un record di update o CLR (anche di transazioni aborted, in pratica faccio redo quasi di tutto), tranne quando:

- la pagina non è in dirty page table(quindi già su disco oppure la transazione non ha fatto commit e posso perdere le modifiche della pagina)
- la pagina e' in dirty page table ma ha recLSN > LSN (la pagina di questa transazione e' stata resa persistente e dopo un'altra transazione ha modificato quella pagina in futuro, reinserendola in dpt con un nuovo recLSN, quindi evito di fare REDO)
- pageLSN > LSN (quindi esiste un record di log che ha un update su quella pagina, evito di rifare il REDO ora quando so che devo già farlo in futuro → evito lavoro inutile. simile al caso sopra)

Al redo della pagina setto pageLSN=LSN
In questa fase i CLR vengono sempre 'redo'

UNDO

Per gli undo vado indietro. Parto che ho un set (toUndo) di lastLSN, uno per ogni transazione crashata non committed che hanno pagine modificate in disco(quelle dentro transaction table):
Prendo il LSN più grande dentro toUndo

- se questo LSN e' un CLR and undonextLSN == null → scrivo END
- se questo LSN e' un CLR and undonextLSN != null → aggiungi undonextLSN al set toUndo
- se questo LSN e' un update → faccio undo dell'update e scrivo un CLR con undonextLSN=prevLSN. Poi inserisco prevLSN al set toUndo

ciclo finché toUndo non e' vuoto. In pratica tutti i CLR non devo rifarli, perchè sono già stati rifatti, però li uso per cercare il primo update che devo rifare, perche i CLR contengono i undonextLSN.
In questa fase i CLR non vengono mai 'undo' di nuovo

FINE