QUERY PROCESSING

Modulo del DBMS che, data una query si occupa di:

1) Parsing e traduzione:
   da SQL a algebra relazionale
   controllo della sintassi
2) Ottimizzazione:
   sceglie l'implementazione più efficiente della query
3) Evaluation engine:
   dato un QEP restituisce la risposta alla query

SELEZIONE	$\sigma_c(R)$	$\sigma_{nome = "andrea"}(student)$
PROIEZIONE	Π lista di attributi (R)	$\Pi_{id, nome}(student)$
PRODOTTO CART	$r \times s$	course $\times$ year
JOIN	$r \bowtie_c s$	student $\bowtie_{id = s\_id}$ exam

OTTIMIZZAZIONE

1° LIVELLO: ordine delle operazioni
2° LIVELLO: implementazione delle operazioni

Sceglie il migliore QEP
1) genera ed esegue "tutti" i QEP usando euristiche
2) sceglie quello con il costo stimato minore

Il costo è calcolato utilizzando il Catalog:
- META INFORMAZIONI: # tuple, dimensioni, # attributi...
- INFO STATISTICHE: stimare il selectivity rate di una condizione

COST MODEL

Non solo le dimensioni dell'input ma anche altri fattori:
- hardware (permanenti)
- risorse all'esecuzione (run time)

È necessario fare una comparazione teorica degli algoritmi

Non si valuta sul tempo ma sul consumo di risorse:

- accesso al disco ← più rilevante, ce ne sono molti
- utilizzo di CPU
- utilizzo della rete ← importanti nei DDBMS

Accesso a disco:

- seeking: posizionare la testa del disco nella giusta posizione
- transfering: dal disco / memoria o memoria / disco

t_S: costo di un movimento della testa del disco

t_T: costo del trasferimento di un blocco

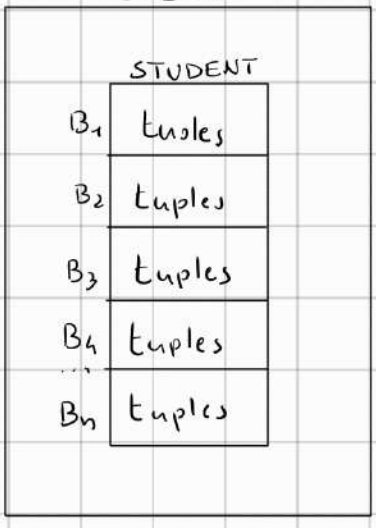
ALGORITMI PER LE OPERAZIONI RELAZIONALI

$\#_S$: numero di operazioni di seeking in un algoritmo

$\#_T$: numero di blocchi trasferiti in un algoritmo

La complessità di un algoritmo è:

$$t_S \cdot \#_S + t_T \cdot \#_T$$



Le relazioni sono salvate nel disco come file fatti di blocchi consecutivi

Le tuple sono ordinate in base ad un attributo

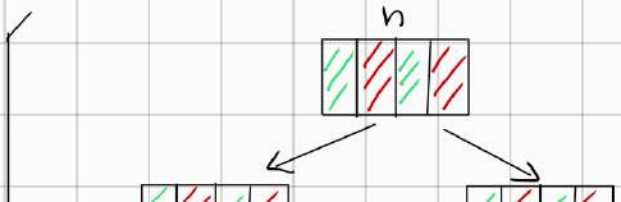
$$\#_S = 1 \quad \#_T = b_R \quad t_S + t_T \cdot b_R$$

b_R: numero di blocchi che compongono la relazione R

Gli algoritmi studiati:

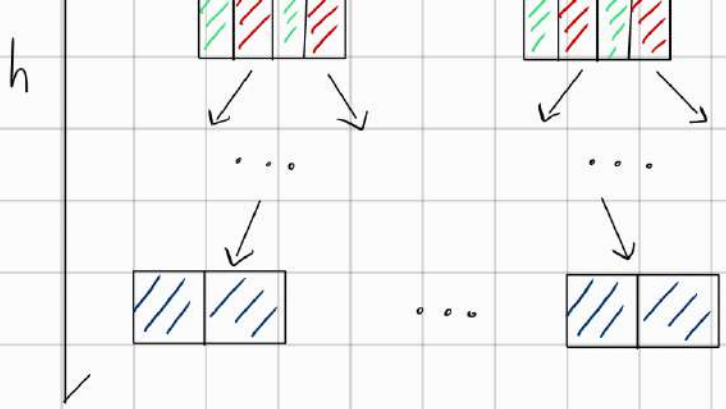
- Index / no index
- Primary / Secondary index
- La condizione ha a che fare con una primary Key
- Condizioni semplici / complesse (=, < ... vs and, or ...)

INDEX (B^+ trees): struttura ad albero bilanciato salvato sul disco



A attributo viene creato un albero

Ogni nodo contiene n coppie



valore / puntatore a nodo successivo

Le foglie contengono i riferimenti ai valori della relazione sul disco

I figli contengono valori compresi tra valore e valore successivo

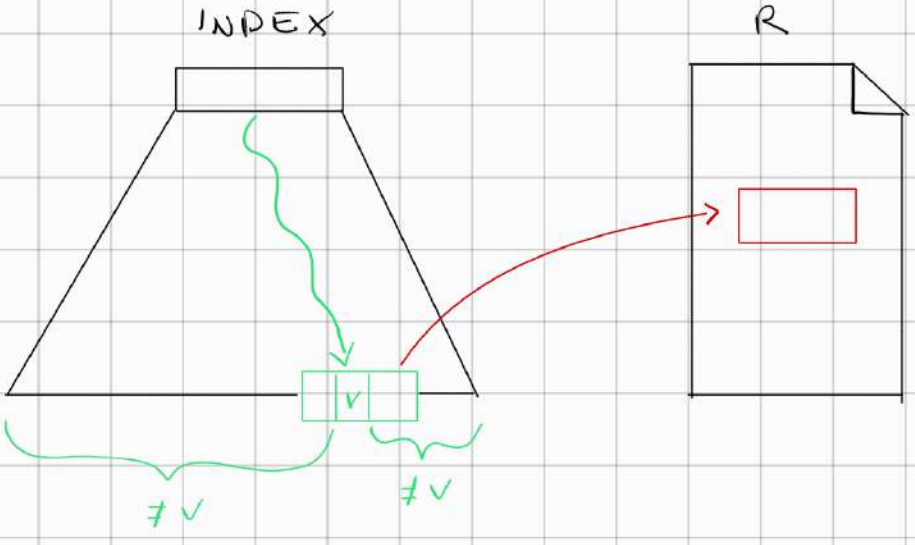
La distanza tra root e qualsiasi foglia è h, di conseguenza:

$\#_S = h \quad \#_T = h$

UGUALIANZA $\sigma_{A=v}(R)$

Vogliamo trovare le tuple di R che soddisfano la condizione

1) Primary index / A è PK



$\#_S = h + 1$

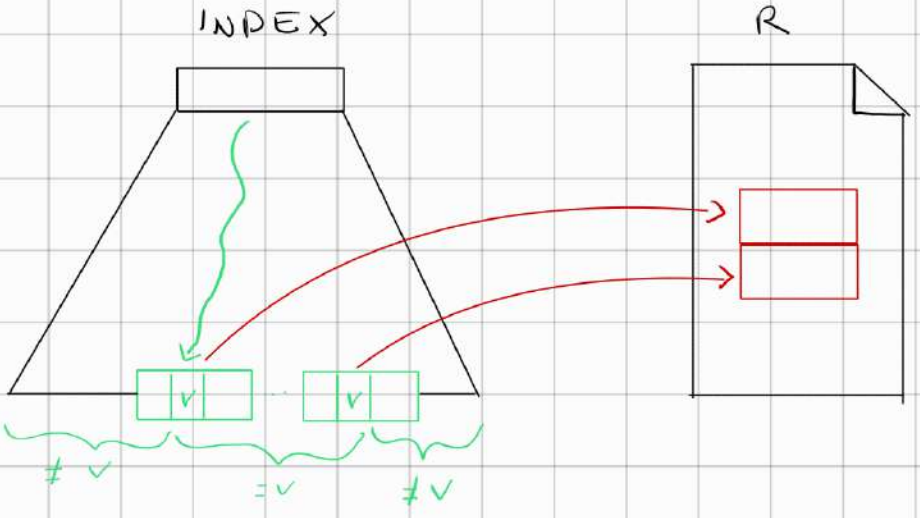
$\#_T = h + 1$

2) Secondary index / A è PK

I blocchi non sono ordinati rispetto agli indici

La complessità non cambia

3) Primary index / A non è pK

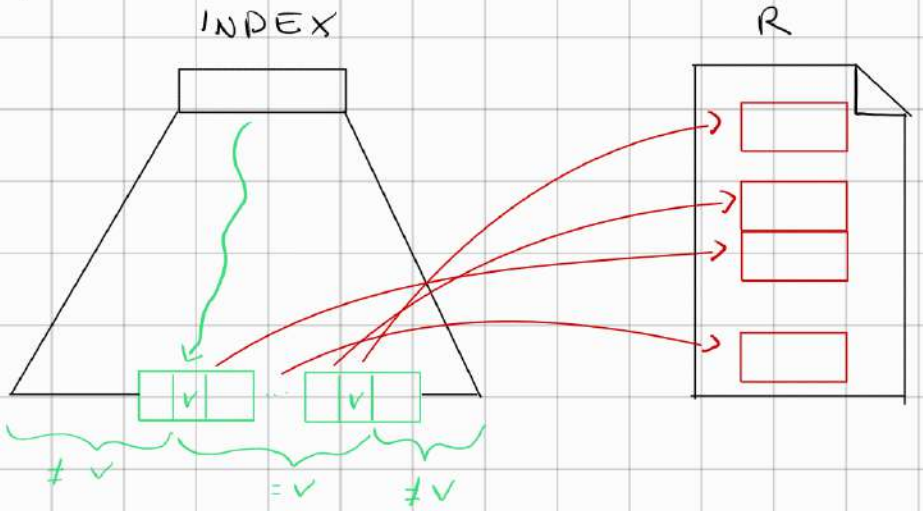


R è contigua in memoria, basta trovare solo il primo

$\#_S = h + 1$

$\#_T = h + b$

4) Secondary index / A non è PK



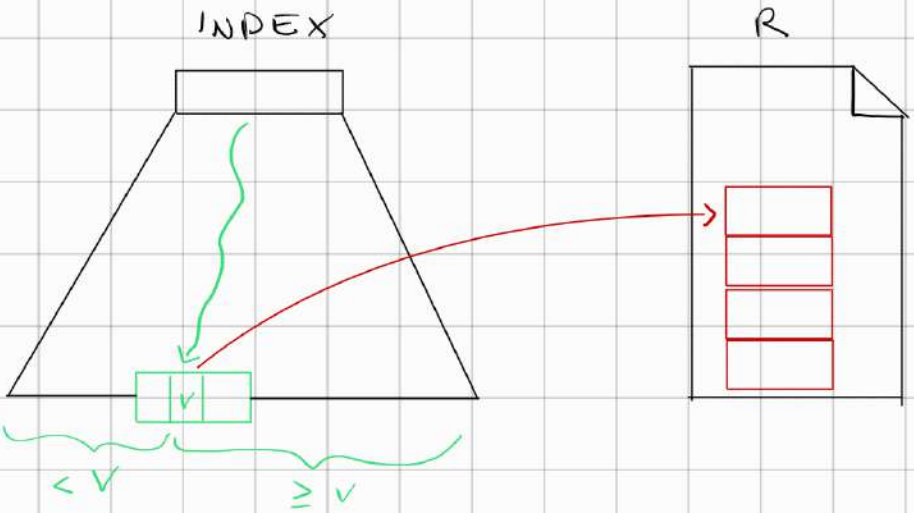
$\#_S = h + n$

$\#_T = h + n$

n è il numero di tuple che combacions

$>, \geq \quad \sigma_{A>v}(R)$

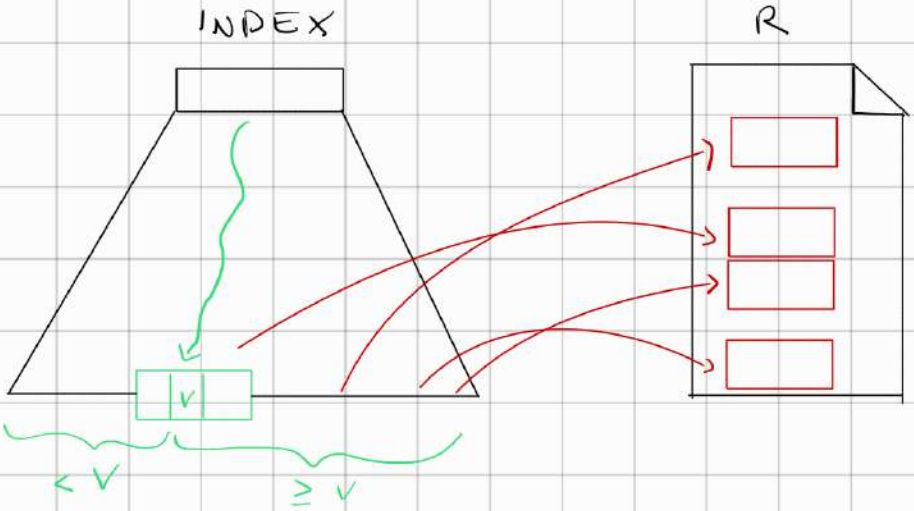
Primary index



$\#_S = h + 1$

$\#_T = h + b$

Secondary index



$\#_S = h + n$

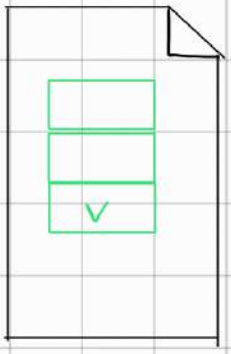
$\#_T = h + n$

$<, \leq \quad \sigma_{A<v}(R)$

Primary index

Non serve usare l'indice, basta un file scan

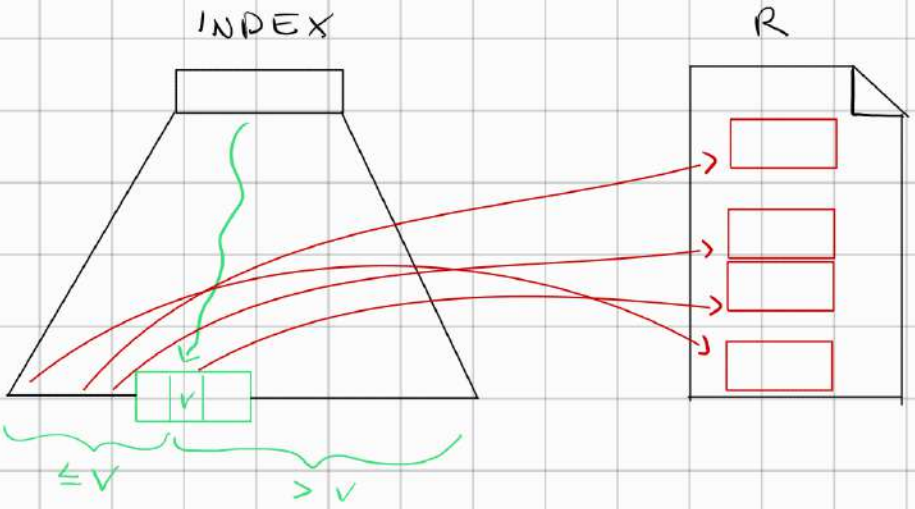
R



$\#_S = 1$

$\#_T = n$

Secondary index



$\#_S = h + n$

$\#_T = h + n$

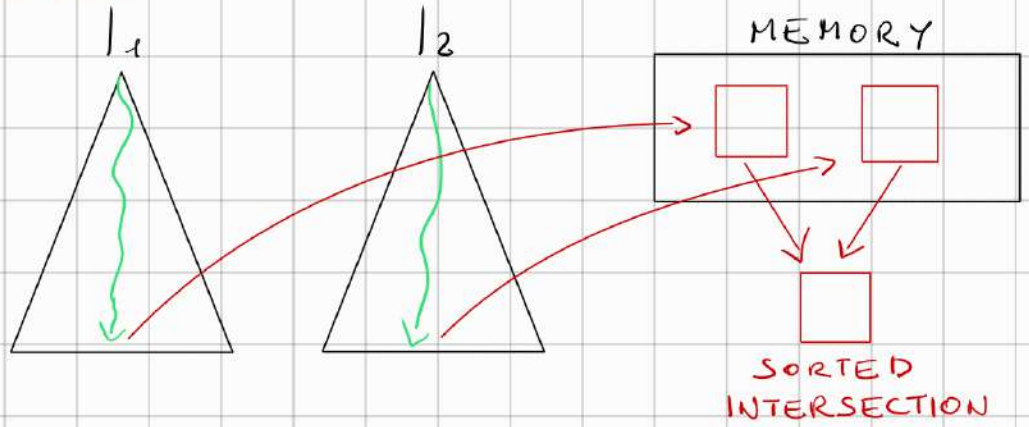
Solitamente fare un file scan è più efficiente di un secondary index in base al selectivity rate = $\frac{\#\text{matching tuples}}{\#\text{tuples}}$ della condizione

CONGIUNZIONE $\sigma_{c_1 \& \dots \& c_n}(R)$

Conosciamo il costo di $\sigma_{c_i}(R)$, $\forall i \in \{1 \dots n\}$

Il costo stimato della query è $\min\{cost(\sigma_{c_i}(R))\}$

Se non sono presenti primary index l'intersezione (ordinata) avviene in memoria



$\#_S = h_1 + h_2 + b_{1\wedge 2}$

$\#_T = h_1 + h_2 + b_{1\wedge 2}$

Funziona se i dati possono stare tutti in memoria

DISGIUNZIONE

Come per la congiunzione, ma si effettua l'unione ordinata

$\#_S = b_1 + b_2 + b$

$\#_T = b_1 + b_2 + b$

NEGAZIONE

$\neg(C_1 \wedge C_2) \equiv \neg C_1 \vee \neg C_1 \dots$

Si può quindi ridurre ad altre operazioni

SORTING

Richiesto dalla query o utilizzato dal DBMS per ottimizzare

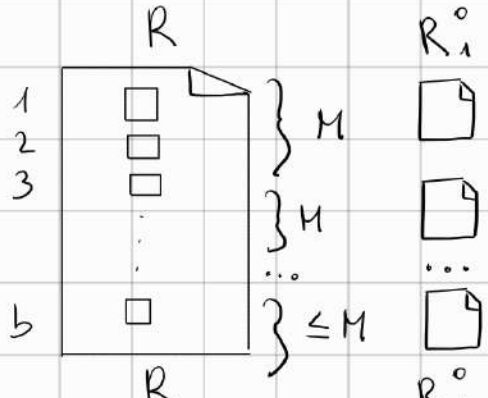
Se R sta tutta in memoria si può usare un alg. di sort classico

EXTERNAL MERGE SORT

Merge: unione ordinata di più insiemi ordinati

1) Vengono create e ordinate $n_0 = \lceil b_R/M \rceil$ runs

2) Ogni run viene salvata in memoria



Ogni blocco è r/w una sola volta

$\#_S = 2\lceil b_R/M \rceil = 2n_0$

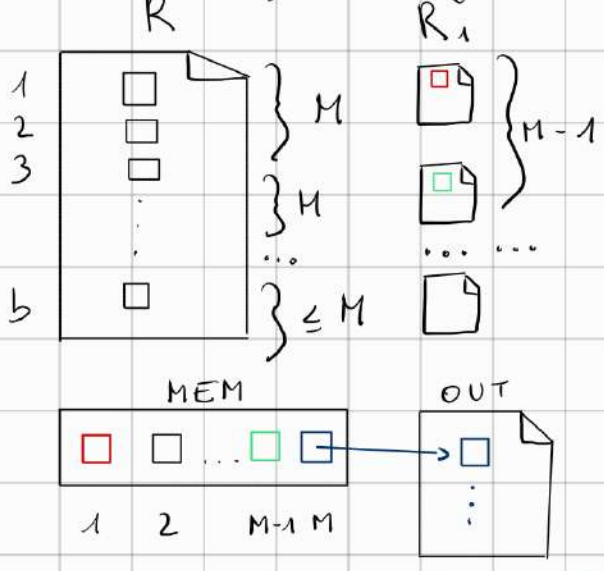
$\#_T = 2b_R$

3) Viene effettuato il merge delle run

Vengono scelte le teste di M-1 file e si effettua il merge

Il blocco M è il blocco di output, nel quale viene scritto il valore minore tra le M-1 teste e scritto nel file di output

Questo processo continua iterativamente



$\#_{STEPS} = \log_{M-1} \lceil b_R/M \rceil$

$\forall$ STEP:

$\#_S = 2b_R$

$\#_T = 2b_R$

OTTIMIZZAZIONE: vengono tenuti K_i blocchi in memoria M_i per ogni file

e $M-K$ per l'output, riducendo il numero di seeking operations

$\#_S = 2\lceil b_R / K \rceil$

$\#_{STEPS} = \lceil \log_{\frac{M}{K-1}} \lceil b_R / M \rceil \rceil$

JOIN $R \underset{R.A=R.S}{\bowtie} S$

L'ordine delle relazioni influenza l'efficienza

NESTED LOOP JOIN

$\forall$ tupla $t_r \in R$ — R è l'outer relation

  $\forall$ tupla $t_s \in S$ — S è l'inner relation

    se (t_r, t_s) soddisfa la condizione del join

      aggiungo $t_r \cdot t_s$ all'output

Se c'è abbastanza memoria per la relazione più piccola:
Si salva quella interna perché viene letta più spesso

$\#_S$: $1+1$

$\#_R$: $b_R + b_S$

C'è abbastanza memoria per solo un blocco per ogni relazione

R	S

Copio un blocco di R e leggo interamente S

- S viene letta n_R volte
  - una lettura di S: $\#_S = 1$ $\#_T = b_S$
- R viene letta una volta

$\#_S = n_R(\cdot 1) + b_R$

$\#_T = n_R \cdot b_S + b_R$

BLOCK NESTED LOOP JOIN

$\forall$ blocco $B_R \in R$

  $\forall$ blocco $B_S \in S$

    $\forall$ tupla $t_r \in B_R$

      $\forall$ tupla $t_s \in B_S$

        se (t_r, t_s) soddisfa la condizione del join

aggiungo $t_R \cdot t_S$ all'output

S viene letta interamente b_R volte anziché n_R

$\#_S = b_R + b_R$

$\#_T = b_R \cdot b_S + b_R$

INDEXED NESTED LOOP JOIN

Se è presente un'indice per una relazione, si può usare come inner rel.

$\forall$ tupla $t_R \in R$

index scan su S per trovare la tupla t_S che soddisfa la condizione

aggiungo $t_R \cdot t_S$ all'output

VALUTAZIONE DELLE ESPRESSIONI

Quando il DBMS deve valutare espressioni complesse usa due tecniche:

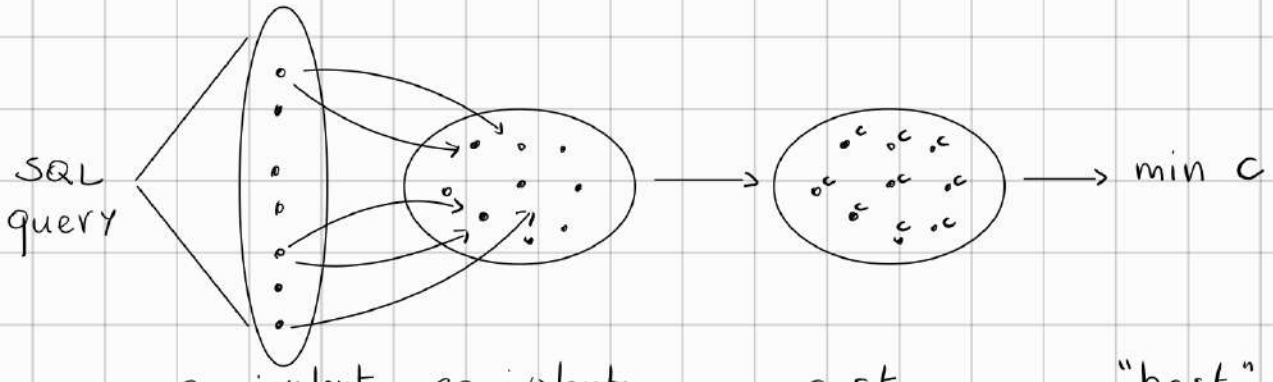
- MATERIALIZZAZIONE: le tuple risultati vengono salvati in relazioni temporanee sul disco. Alto uso di memoria ma sempre possibile
- PIPELINING: le tuple vengono passate direttamente all'operazione successiva. Meno uso del disco ma non sempre possibile

La materializzazione viene usata nella costruzione dell'indice

Il pipelining permette la parallelizzazione delle operazioni

- le operazioni bloccanti (proiezione, sort ...) necessitano di materializzazione

QUERY OPTIMIZATION



EQUIVALENCE RULES

Due espressioni in AR sono equivalenti se generano lo stesso set di tuple

Una regola di equivalenza afferma che due espressioni sono equivalenti

1) Selezioni sequenziali possono essere unite
2) La selezione è commutativa
3) In una serie di proiezioni importa solo la più esterna
4) La selezione può essere combinata con il prodotto cartesiano e θ join
   a. $\sigma_\theta(R \times S) \equiv R \bowtie_\theta S$
   b. $\sigma_{\theta_1}(R \bowtie_{\theta_2} S) \equiv R \bowtie_{\theta_1 \wedge \theta_2} S$
5) I join sono commutativi, l'ordine è importante per l'efficienza
6) I join sono associativi
$$(R \bowtie S) \bowtie T \equiv R \bowtie (S \bowtie T)$$
$$(R \bowtie_{\theta_1} S) \bowtie_{\theta_2 \wedge \theta_3} T \equiv R \bowtie_{\theta_1 \wedge \theta_3} (S \bowtie_{\theta_2} T)$$
7) La selezione si distribuisce sul join
$$\sigma_{\theta_1}(R \bowtie S) \equiv \sigma_{\theta_1}(R) \bowtie S$$
$$\sigma_{\theta_1 \wedge \theta_2}(R \bowtie S) \equiv \sigma_{\theta_1}(R) \bowtie \sigma_{\theta_2}(S)$$

L'ottimizzatore usa insiemi di regole minimi (search space minimo)
E' un problema NP-HARD, si usano quindi euristiche
- statistiche
- teoriche
- greedy

STATSTICAL INFORMATION FOR COST ESTIMATION

Sono contenute nel catalog, una relazione salvata sul DB
- n_R
- b_R
- l_R: dimensioni in byte di una tupla
- f_R: blocking factor, numero di tuple per blocco
- $V(A, R)$: numero di valori distinti dell'attributo A
- $min(A, R)$ e $max(A, R)$
- info sull'indice: altezza, numero di foglie...

Il catalog viene aggiornato periodicamente

SELECTION SIZE ESTIMATION

Numero di tuple che soddisfano la condizione di selezione
- $\sigma_{V=A}(R)$
  - $n_R / V(A, R)$

- 1 se A è pK
- $\sigma_{V \subseteq A}(R)$
  - 0 se $\min(A,R) > V$
  - n_R se $\max(A,R) \leq V$
  - $n_R \cdot \frac{V - \min(A,R)}{\max(A,R) - \min(A,R)}$ altrimenti

Se nel catalog è presente un'istogramma per A, V cadrà in una sola classe e considero la sua numerosità

COMPLEX SELECTION SIZE ESTIMATION

- $E = \sigma_{\theta_1 \wedge \dots \wedge \theta_n}(R)$ $\quad n_R \cdot SR(E) = n_R \cdot \frac{S_1 \cdot \dots \cdot S_n}{n_R^n}$
  dove S_i è la dimensione di $\sigma_{\theta_i}(R)$
- $E = \sigma_{\theta_1 \vee \dots \vee \theta_n}(R)$ $\quad SR(E) = 1 - SR(\sigma_{\neg\theta_1 \wedge \dots \wedge \neg\theta_2}(R))$
  $n_R \cdot \left[1 - \left(1 - \frac{S_1}{n_R}\right) \cdot \dots \cdot \left(1 - \frac{S_n}{n_R}\right)\right]$
- $E = \sigma_{\neg\theta}(R)$ $\quad n_R - n_{(\sigma_\theta(R))}$

JOIN SIZE ESTIMATION

- $R \times S$ $\quad n_R \cdot n_S$
- $R \bowtie_A S$ $\quad n_R \cdot n_S / \max\{V(A,R), V(A,S)\}$
- $R \bowtie_\theta S = \sigma_\theta(R \times S)$ prodotto cartesiano e selezione

PROJECTION SIZE ESTIMATION

$E = \Pi_A(R)$

$n_E = n_R$ con duplicazioni

$V(A,R)$ senza duplicazioni

SCELTA DEL QEP

Scegliere il miglior algoritmo per ogni singola operazione può non portare alla soluzione ottimale

Un ottimizzatore incorpora due approcci:

- esplora lo spazio dei QEP e sceglie quello con costo minore
- Sceglie il piano in base ad delle euristiche

COST-BASED OPTIMIZATION

Trova il QEP più efficiente in base:

- statistiche riguardanti le tabelle (# righe, indici, distribuzioni ...)
- genera diversi piani di esecuzione (operazioni, ordine, algoritmi ...)
- ottimizza in base ai costi stimati (CPU, I/O, memoria)

Scegliere l'ordine ottimo delle operazioni di join

$(2(n-1))!/(n-1)!$ ordini

es. $r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4 \bowtie r_5$ 1680 ordini

- calcolo il miglior ordine una volta $r_1 \bowtie r_2 \bowtie r_3$ 12 ordini
- calcolo il miglior ordine per $r_{123} \bowtie r_4 \bowtie r_5$ 12 ordini

HEURISTIC OPTIMIZATION

L'ottimizzazione basata sul costo è completa, ma si possono usare delle euristiche per diminuire lo spazio delle soluzioni

Le euristiche trasformano il query tree utilizzando regole che solitamente migliorano le performance:

- dare precedenza alla selezione (riduce il numero di tuple)
- dare precedenza alla proiezione (riduce il numero di attributi)
- eseguire prima la selezione o join con condizione più restrittiva, ovvero con dimensione del risultato più piccola
- preferire i left deep join, interagiscono meglio con gli algoritmi

DISTRIBUTED DBMS

COMPUTAZIONE DISTRIBUITA: insieme di elementi di processazione autonomi interconnessi da una rete e che cooperano per eseguire un compito

DISTRIBUTED DATABASE SYSTEM: DDB + DDBMS

Sia i dati che l'algoritmo che ne controlla la consistenza, l'accesso concorrente e la robustezza del sistema sono distribuiti

PROPRIETÀ:

- TRASPARENZA: separazione dei livelli più alti da quelli inferiori
- AFFIDABILITÀ: la duplicazione e distribuzione dei dati (non c'è un singolo punto di fallimento)
- PERFORMANCE: replicazione garantisce la data localization. La distribuzione favorisce il parallelismo
- ESPANSIONE

PROBLEMATICHE:

- COMPLESSITÁ
- SINCRONIZZAZIONE
- REPLICAZIONE e DISTRIBUZIONE
- CAP THEOREM: é impossibile per un DS garantire contemporaneamente:
  - CONSISTENCY: dati aggiornati e corretti
  - AVAILABILITY: ogni richiesta viene servita
  - PARTITION TOLLERANCE: tolleranza a fallimenti di comunicazione

DDBMS DESIGN

Come distribuire i nodi sulla rete:

- come frammentare i dati
- come allocare i frammenti nei veri nodi
- garantire la correttezza dalla frammentazione e allocazione

FRAMMENTAZIONE

Dividere i dati per ottimizzare l'accesso (LOCALITY) e migliorare le performance (INTRA-QUERY PARALLELISM)

- ORIZZONTALE (HF): suddividere la relazione in base ai valori degli attributi
  - PRIMARIA
  - DERIVATA
- VERTICALE: suddividere la relazione in sotto-relazioni contenenti solo alcuni attributi
- IBRIDA

CORRETTEZZA

- COMPLETEZZA: ogni dato deve essere presente in almeno un frammento
- RICOSTRUIBILITÁ: deve esistere un'operatore relazionale che ricostruisca la relazione originale combinando i frammenti
- DISGIUNZIONE: ogni frammento deve contenere dati distinti

PHF

La suddivisione avviene attraverso predicati semplici (PARETO)
La relazione é suddivisa in frammenti che contengono tuple che

soddisfano uno specifico minterm

minterm: combinazione di predicati. Ogni predicato può essere incluso o escluso dal minterm in modo da coprire tutte le combinazioni di frammentazione

L'insieme dei predicati semplici deve essere COMPLETO e MINIMO, ovvero che ogni Q accede a tutte o a nessuna tupla di un frammento

- COMPLETEZZA: la definizione dei minterms garantisca completezza
- RICOSTRUIBILITÀ: unione dei frammenti
- DISGIUNZIONE: i minterm sono costruiti in modo mutualmente esclusivo

DHF

La frammentazione avviene da una relazione padrona alla quale una relazione membro è collegata tramite chiave esterna

Ottimizza l'accesso ai dati correlati

1) Si frammenta il proprietario in base ai predicati
2) Si applica la frammentazione derivata al membro, in modo che le tuple associate a una specifica chiave del proprietario siano nello stesso frammento

SEMI JOIN

$R \ltimes S \subseteq R \equiv \Pi_{V\ attr\_R}(R \bowtie S)$ join che restituisce solo le tuple della outer relation

Questo operatore viene utilizzato per propagare la frammentazione

Una relazione proprietaria S frammentata in $F_S = \{S_1 \dots S_n\}$ i frammenti derivati della relazione membro R saranno:

$$R_i = R \ltimes S_i$$

- COMPLETEZZA: garantita dai vincoli di integrità della FK
- RICOSTRUIBILITÀ: unione dei frammenti
- DISGIUNZIONE: garantita dai vincoli di integrità della FK

VF

Ogni frammento contiene un sotto insieme di attributi

Utilizzando la QUERY accedono sempre a sotto insiemi di attributi

Ogni frammento deve contenere la PK della relazione di appartenenza
Viene generata una matrice di utilizzo degli attributi che memorizza a quali attributi accedono le query
Viene misurata l'affinità tra gli attributi

$$aff(A_i, A_j) = \sum_q use(A_i, q) \cdot use(A_j, q) \cdot acc(q)$$

dove $acc(q)$ è la frequenza della query q
Utilizzando l'affinità gli attributi vengono frammentati
- COMPLETEZZA: garantita dall'algoritmo di partizione
- RICOSTRUIBILITÀ: join sulla PK dei frammenti
- DISGIUNZIONE: garantita dall'algoritmo di partizione

HYF
1) Scegliere il primo tipo di frammentazione
   - HF se le query accedono spesso a sottoinsiemi di tuple
   - VF se le query accedono spesso a sottoinsiemi di attributi
2) Frammentare i frammenti utilizzando la tecnica opposta

La relazione può essere ricostruita applicando a ritroso i metodi di ricostruzione delle varie tecniche di frammentazione

DATA DIRECTORY (CATALOG)
Oltre ai metadata del contesto centralizzato contiene:
- info sulla ricostruzione dell'intero DB
- in quale nodo sono memorizzate relazioni e frammenti

Anch'esso deve essere frammentato e allocato correttamente

DISTRIBUTED QUERY PROCESSING

DATA LOCALIZATION - 1ª E 2ª FASE
Trasforma una query globale applicata a relazioni distribuite in una query localizzata applicata ai frammenti
Vengono utilizzate informazioni globali sulla distribuzione dei frammenti

DATA LOCALIZATION ALGORITHM
Una query localizzata è ottenuta da una query globale sostituendo le relazioni con il rispettivo localization program, espressione in AR

$(\cup/\bowtie)$ che ricostruisce la relazione dai suoi frammenti invertendo le regole di frammentazione

Questo algoritmo non è efficiente, avviene quindi una prima fase di ottimizzazione, chiamata REDUCTION:

- PHF: se il predicato di frammentazione è contraddittorio con quello di selezione / join, il frammento viene ignorato, non sempre è conveniente. Distribuire il join sulle unioni:

$$R \bowtie S \equiv (R_1 \cup R_2) \bowtie (S_1 \cup S_2) \equiv (R_1 \bowtie S_1) \cup (R_2 \bowtie S_2)$$

- DHF: stessa tecnica + il join può essere processato sui frammenti in parallelo
- VF: si può ignorare la proiezione quando l'insieme degli attributi della query $\cap$ dell'insieme di attributi del frammento include la PK

DISTRIBUTED QUERY OPTIMIZATION - 3ª FASE

Produrre un DQEP, simile a quello centralizzato, ma deve prevedere anche:

- SEARCH SPACE: deve includere i metodi di comunicazione
- COST MODEL: deve includere i costi di comunicazione

ORDINE DEI JOIN

È fondamentale nel contesto distribuito per minimizzare i costi di comunicazione

- 2 RELAZIONI: assumiamo che sia già avvenuta la localizzazione
  - spostare la relazione più piccola e usarla come outer perché
    - l'indice sulla più grande non viene perso nell' inlj
    - non serve salvarla nel nlj o bnlj
- M RELAZIONI: spostare le relazioni più piccole

SEMI-JOIN ALGORITHM

Usa il semi-join per ridurre il numero di tuple da trasferire

$$R \bowtie S \equiv (R \ltimes S) \bowtie S \equiv (R \ltimes S) \bowtie (S \ltimes R)$$

È conveniente se $size(R \ltimes_A S) << size(R \bowtie_A S)$ a costo di $\Pi_A(R)$

Usare il semi-join fa perdere l'indice

TRANSFER

Nel contesto distribuito ci sono:

- il sito master dove la query è inizializzata. Sceglie il sito di esecuzione e il metodo di trasferimento
- i siti apprendista dove i frammenti sono salvati e le query eseguite. Ottimizzano le query parziali a loro assegnate

$R \bowtie S$ può essere eseguita:
- dove R è salvata
- dove S è salvata
- in un terzo sito (permette il trasferimento parallelo)

METODI DI TRASFERIMENTO
- SHIP WHOLE: tutta la relazione viene trasferita
- FETCH AS NEEDED: solo le tuple necessarie (semi-join)

TRANSACTION MANAGER

Una transazione è un'insieme di operazioni che garantiscono il passaggio consistente dei dati da uno stato all'altro

Proprietà ACID, garantiscono la validità dei dati in caso di errore:
- ATOMICITÀ: ogni transazione è indivisibile. Viene eseguita completamente oppure non viene eseguita affatto
- CONSISTENZA: rispetta i vincoli di integrità
- ISOLAMENTO: le modifiche non sono visibili fino al commit
- DURABILITÀ: una volta eseguito il commit la modifica rimane valida anche in caso di errori e non può essere modificata

CONDIZIONI DI TERMINAZIONE
- COMMIT: la transazione ha successo, il DB passa ad un nuovo stato visibile a tutti
- ABORT: la transazione ha fallito, viene effettuato il ROLLBACK

CONCORRENZA

Gestisce l'esecuzione simultanea di più transazioni
- SERIALIZZABILITÀ: l'ordine di esecuzione delle transazioni è seriale

(DISTRIBUTED) CONCURRENCY CONTROL

Le transazioni possono essere modellate come insiemi parzialmente ordinati di operazioni di lettura/scrittura e condizioni di terminazione. Le operazioni in conflitto (agiscono sullo stesso dato) vanno ordinate

STORIE SERIALI

Se l'insieme delle operazioni di tutte le transazioni non si intersecano. Garantiscono il mantenimento della consistenza del DB

LOCKING ALGORITHM

Generano una storia serializzabile. Le richieste di scrittura / lettura sono gestite da uno scheduler che decide a chi garantire il lock

- 2PL: una risorsa viene bloccata prima di essere usata da una transazione. Una volta rilasciata, non può essere richiesto un altro lock
- Strict 2PL: le transazioni mantengono i lock fino al commit. Aumenta l'isolamento ma diminuisce la concorrenza

D2PL

Ogni sito ha uno scheduler 2PL che gestisce le richieste per i propri dati

DEADLOCK

Si verifica quando due o più transazioni attendono indefinitamente che l'altra rilasci il lock di una risorsa

- PREVENZIONE: impedire azioni rischiose
- EVITAMENTO: identificare potenziali DL e intervenire
- RECUPERO: rilevare i DL e rimuoverli utilizzando il grafo wait-for

DDLM

I WFG locali vengono inviati ai vari siti per rilevare cicli globali

DATA WAREHOUSING

Processo di raccolta e organizzazione di grandi quantità di dati per supportare l'analisi e il processo decisionale. È un sistema centralizzato per conservare dati e semplificarne l'accesso

DB VS DW

- i DB conservano dati transazionali, le DW dati storici
- i DB sono normalizzati per minimizzare la ridondanza e ottimizzare le operazioni di modifica, le DW sono denormalizzati per ottimizzare le query analitiche a discapito delle operazioni di modifica
- i DB vengono aggiornati frequentemente, le DW periodicamente

OLTP (Online Transaction Processing)

Gestiscono operazioni transazionali efficientemente (inserimento, aggiornamento e cancellazione)

- operazioni frequenti e di piccole dimensioni
- dati normalizzati
- tempi di risposta brevi
- integrità (ACID)

OLAP (Online Analytical Processing)

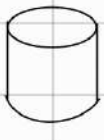
Gestire analisi su enormi quantità di dati storici

- operazioni complesse su molti dati
- dati denormalizzati
- ottimizzati per rispondere velocemente a query su dati aggregati
- bassa frequenza di aggiornamento

DW ARCHITECTURE

- SEPARAZIONE: tra parte analitica e transazionale
- SCALABILITÁ: HW facilmente migliorabile man mano che i dati aumentano
- ESTENDIBILITÁ: possibilità di implementare nuove applicazioni senza modificare il sistema

SINGLE LAYER ARCHITECTURE



DATA SOURCES: operational + external sources

↓

VDW

VIRTUAL DW: serie di algoritmi e query che ristrutturano i dati. I dati sono sempre memorizzati nelle data source

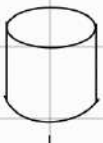
↓

ANALYST

È un layer che ottimizza le query analitiche

Non c'è separazione, si usa se le necessità di analisi dell'azienda sono limitate

TWO LAYER ARCHITECTURE



DATA SOURCES: operational + external sources

↓

ETL

Extract: algoritmi che accedono le varie source

Tranform: migliorare la qualità dei dati. Non esiste un

↓ metodo standard, dipende dal dominio

DW — Load: adattare i dati al modello multidimensionale

↙ ↓ ↘

DMs — DATA MART (opzionali): quando i dati sono troppi vengono divisi in sottoinsiemi in base al dipartimento / categoria migliorando le performance

OLAP TOOLS:

- REPORTING
- DATA MINING — Strumenti per creare visualizzazioni dinamiche, report,
- WHAT IF — Fare predizioni...

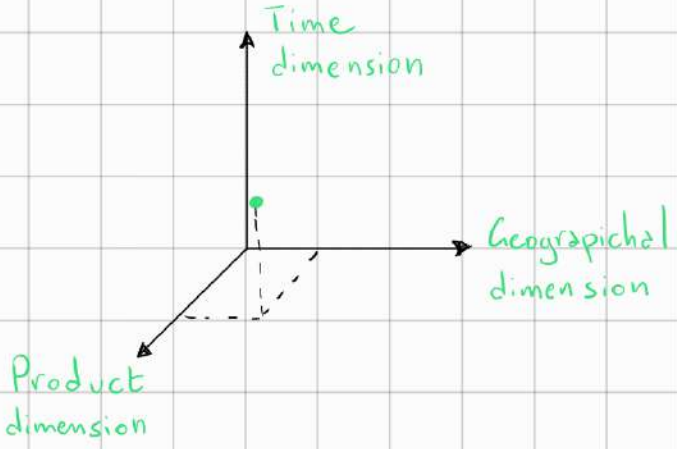
THREE LAYER ARCHITECTURE

Tra le data source e il modulo ETL si pone un RECONCILED DB, un DB relazionale, che si occupa di riunire i dati dalle source. Rappresenta un'ulteriore livello di separazione

MULTIDIMENSIONAL MODEL

- Fact: aspetto importante per un'azienda per il quale vuole un'analisi
- Event: istanza di un Fatto
- Measures: informazioni quantitative che descrivono un Fatto
- Dimension: descrivono il contesto dei fatti (es. tempo, paese...)

I fatti sono rappresentati come CUBI, gli eventi come CELLE



Le dimensioni hanno diversi livelli di aggregazione, detti gerarchia di roll up, che permette l'analisi su vari livelli di dettaglio

AGGREGAZIONE: permette di raggruppare i dati secondo le gerarchie di roll up per ogni dimensione, permettendo vari livelli di granularità e ottimizzare le query OLAP

MULTIDIMENSIONAL MODEL IMPLEMENTATION

MOLAP

Multidimensional OLAP, i dati sono salvati fisicamente in cubi (md arrays)

- più veloce, accesso diretto ai dati
- genera un cubo sparso ~ 80% non usato
- no standardization (implementazioni private)

ROLAP

Relational OLAP, la multidimensionalità viene simulata da relazioni

- più lento, accesso sequenziale ai dati

HOLAP

Hybrid OLAP, le parti dense sono implementate con MOLAP, le parti sparse in ROLAP

- parti spesso richieste sono implementate in MOLAP
- fatti primari (prima rollup) in MOLAP, fatti secondari (dopo rollup) in ROLAP

DIMENSIONAL FACT MODEL

Modellazione concettuale per rappresentare il modello multidimensionale. Struttura i dati attorno a fatti e dimensioni e ne specifica le relazioni (operazioni)

ROLAP usa uno schema a stella con al centro il fatto e i vari nodi sono le dimensioni e vari livelli della gerarchia

- vengono creati delle PK artificiali sulle dimensioni per ottimizzare l'utilizzo della memoria
- vengono creati indici sulle FK
- possono allo snowflake schema normalizzando parzialmente le dimensioni

RELIABILITY

Come mantenere atomicità e durabilità nelle transazioni

Strategie di update:

- IN PLACE: la modifica cambia uno o più valori nel DB
  Efficiente, difficile da undo/redo
- OUT OF PLACE: la modifica crea nuovi valori nel DB
  Non efficiente, facile da undo/redo

Ogni transazione, oltre alle operazioni, deve aggiornare il file di log che contiene le info per fare il recovery

- transaction id
- operazioni eseguite
- valori prima della modifica (before image)
- valori dopo la modifica (after image)

WRITE AHEAD LOG

Nel caso il fallimento avvenga prima dell'aggiornamento del log

- prima di aggiornare il DB, modifica la parte di undo
- prima di fare il commit, modifica la parte di redo

Si usano checkpoints (stati consistenti del DB) per determinare quali azioni undo/redo fare in caso di fallimento

2PC

Protocollo di reliability distribuito

- COORDINATORE: sito da dove è partita la transazione e che gestisce l'esecuzione
- PARTECIPANTI: siti che partecipano all'esecuzione della transazione

1) il coordinatore raccoglie le risposte di commit
2) il coordinatore decide se fare global commit/abort e comunica la sua decisione. Se almeno un sito vota per l'abort, il coordinatore deve fare abort della transazione