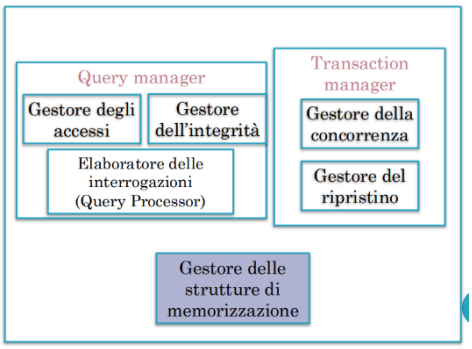
BASI DI DATI (parte della Catania)

Componenti di un DBMS



Tutte le diverse componenti hanno bisogno di accedere ai dati utente e/o dati di sistema per funzionare.
I **dati di sistema** vengono memorizzati nella base di dati insieme ai dati utente; sono gestiti dal sistema come dati utente, disponibili a livello logico come tabelle e memorizzati su disco come file.

Architettura Client-Server

- I servizi interni ed esterni vengono forniti da una singola macchina, il server;
- Le applicazioni risiedono e vengono eseguite su macchine client, dalle quali vengono richiesti i servizi esterni.

Architettura dei DBMS

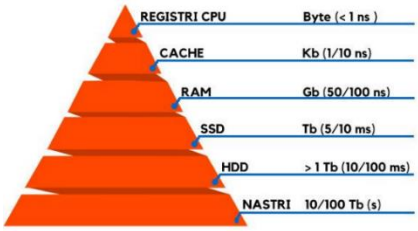
1) Gestore delle strutture di memorizzazione

Le memorie si possono classificare in base alla loro capacità e al loro tempo per trovare certi dati.

Prestazioni di una memoria:

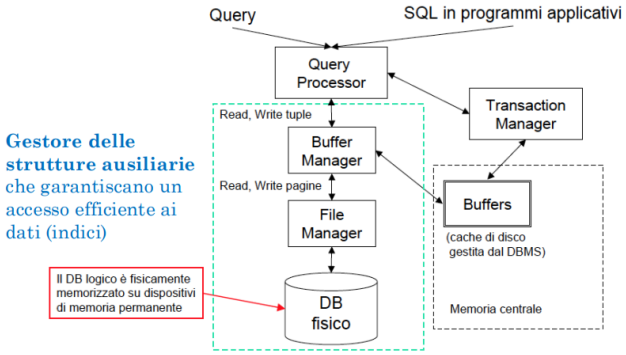
Tempo di accesso ottenuto dalla somma di:

- Latenza: tempo necessario per accedere al primo byte di interesse;
- Tempo di trasferimento: tempo necessario per spostare i dati tra i livelli della gerarchia.



$$\text{tempo di accesso} = \text{latenza} + \frac{\text{dimensione dati da trasferire}}{\text{velocità di trasferimento}}$$

I dati devono essere trasferiti da memoria non volatile a memoria centrale (RAM, cache) per essere elaborati dal DBMS (trasferimento a blocchi/pagine).



Tipi di dispositivi:

Per il loro interesse da un punto di vista storico e la loro diffusione, presentiamo i **dischi magnetici**, consapevoli che altre tecnologie possono comunque essere utilizzate (es. SSD). **Molte funzionalità dei DBMS discendono da caratteristiche dei dischi magnetici.** L'uso intensivo di SSD può richiedere cambiamenti agli approcci tipici utilizzati dai DBMS per fornire i servizi interni. È comunque importante studiare queste soluzioni perché costruiscono i «mattoni» che compongono un qualunque sistema di gestione dati, anche non relazionale.

Hard disk: Un hard disk (HD) è un dispositivo elettro-meccanico per la conservazione di informazioni sotto forma magnetica, su supporto rotante a forma di piatto su cui agiscono delle testine di lettura/scrittura. L'informazione memorizzata su una superficie del disco in cerchi concentrici di piccola ampiezza, detti tracce, suddivise in settori (spesso 512 kb). Per leggere/scrivere un settore si usa un braccio con testina magnetica: il braccio si posiziona sulla traccia e i dati vengono letti/scritti quando il settore passa sotto la testina. Nei dischi a piatti multipli si usa una testina per ogni superficie, su un unico braccio. Le tracce con lo stesso diametro sulle varie superfici sono dette cilindro: i dati sullo stesso cilindro possono essere recuperati molto più velocemente dei dati distribuiti su diversi cilindri.

- **Tempo di latenza:** tempo impiegato per accedere al primo byte di interesse (da 2,8 msec a circa 30 msec, qualche decina di msec nel caso peggiore);
- **Command Overhead Time:** tempo necessario a impartire comandi al drive (trascurabile);
- **Seek Time (Ts):** tempo impiegato dal braccio a posizionarsi sulla traccia desiderata (min 2-10 msec, max 2-20 msec);
- **Settle Time:** tempo richiesto per la stabilizzazione del braccio (trascurabile);
- **Rotational Latency (Tr):** tempo di attesa del primo settore da leggere (da 2 a 11 msec);
- **Tempo di trasferimento:** velocità con cui si trasferiscono byte dai (sui) piatti sulla (dalla) cache del controller. Dipende dalla velocità di trasferimento;
- **Velocità di trasferimento (o transfer rate Tr):** numero di byte trasferiti nell'unità di tempo.

I dati sono trasferiti tra il disco e la memoria principale in unità chiamate blocchi/pagine. Un blocco (o pagina) è una sequenza contigua di settori su una traccia, e costituisce l'unità di I/O per il trasferimento di dati da/per la memoria principale.
Il **tempo di trasferimento di un blocco (Tt)** è il tempo impiegato dalla testina per trasferire un blocco nel buffer, una volta posizionata all'inizio del blocco; tale tempo è molto più breve del tempo di seek e dipende dalla dimensione del blocco (P) e dalla velocità di trasferimento (Tr):

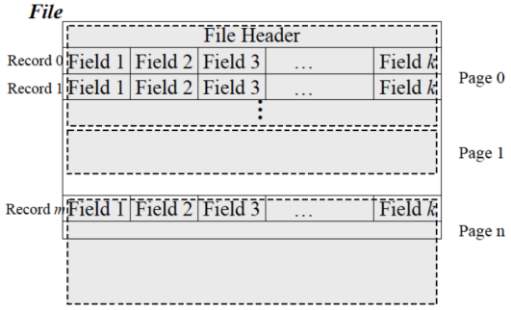
$$Tt = P / Tr$$

Il gestore delle strutture di memorizzazione deve cercare di ridurre il tempo di latenza.

HDD e SSD (Solid State Drive): Nei SSD, non ci sono componenti meccaniche ma solo circuiti elettronici. Le Operazioni di lettura/scrittura generalmente più veloci, Il tempo di accesso inferiore a 10 msec e si riduce il gap tra memoria volatile e non volatile in termini di tempo. Ma gli SSD sono più costosi degli HDD e quindi in genere di dimensione inferiore. Limitano alcune esigenze «storiche» dei DBMS (gestione del buffer, uso di strutture ausiliarie di accesso).

Il DB fisico (livello fisico): A livello fisico, un DB consiste di un insieme di file, ognuno dei quali viene visto come una collezione di pagine/blocchi, di dimensione fissa. Ogni pagina/blocco memorizza più record (corrispondenti alle tuple dell'istanza della base di dati), e, a sua volta, un record consiste di più campi, di lunghezza fissa e/o variabile, che rappresentano gli attributi (le 'colonne' della tabella). Ogni relazione del DB è memorizzata in un proprio file, e tutto il DB è memorizzato in un singolo file.

Organizzazione dei dati nei file:

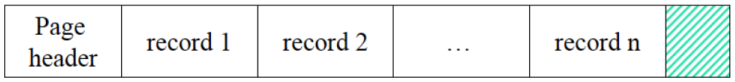


Record: I dati sono generalmente memorizzati in forma di record. Un record è costituito da un insieme di valori collegati, ognuno formato da uno o più byte e corrisponde ad un campo del record. Una collezione di nomi di campi a cui sono associati i tipi corrispondenti costituisce un tipo di record. Per ogni tipo di record nel DB deve essere definito uno schema (fisico) che permetta di interpretare correttamente il significato dei byte che costituiscono il record. Quindi esiste un mapping tra lo schema delle tuple della relazione (schema logico) e un tipo record su file (schema fisico).
In generale ogni record include un **header** che, oltre alla lunghezza del record, può contenere:

- L'identificatore della relazione cui il record appartiene;
- L'identificatore univoco RID del record nel DB;
- Un timestamp che indica quando il record è stato inserito o modificato l'ultima volta;
- Il formato specifico dell'header ovviamente varia da un DBMS all'altro.

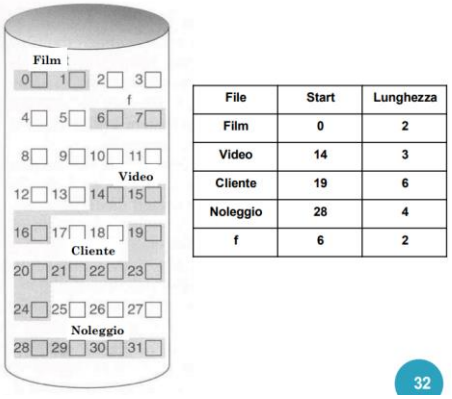
File: un file è una sequenza di record. un file è detto file con record a lunghezza fissa se tutti i record memorizzati nel file hanno la stessa dimensione (in byte), altrimenti, parliamo di file con record a lunghezza variabile.

Organizzazione dei record in blocchi:
Normalmente, la dimensione di un record è molto minore di quella di una pagina. Nel caso di record a lunghezza fissa, l'organizzazione di una pagina si potrebbe presentare così:



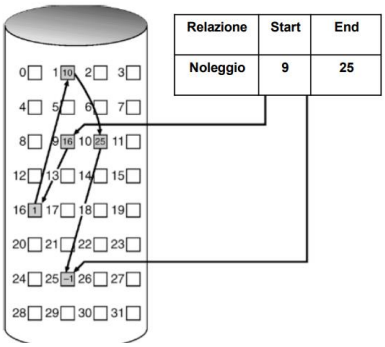
Il page header contiene informazioni quali: ID della pagina del DB, timestamp che indica quando la pagina è stata modificata l'ultima volta, relazione cui le tuple nella pagina appartengono, ecc.
Normalmente, un record è contenuto interamente in una pagina, quindi si può avere spazio sprecato.

Organizzazione dei blocchi su disco:
Allocazione contigua: i blocchi del file sono allocati in blocchi di disco contigui.



File	Start	Lunghezza
Film	0	2
Video	14	3
Cliente	19	6
Noleggio	28	4
f	6	2

Allocazione concatenata: ogni blocco di un file contiene un puntatore al successivo blocco del file. Utilizzo di bucket (ossia un insieme di blocchi, non necessariamente contigui ma 'vicini', possibilmente nello stesso cilindro) per gruppi di record tra loro collegati (ad esempio, tutti i noleggi di un certo cliente).



Organizzazione dei record nei file:
I file che contengono i record dei dati costituiscono l'organizzazione primaria dei dati. Il modo con cui i record vengono organizzati nei file incide sull'efficienza delle operazioni e sull'occupazione di memoria. Poiché i dati sono trasferiti in blocchi tra la memoria secondaria e la memoria principale, è importante assegnare i record ai blocchi in modo tale che uno stesso blocco contenga record tra loro interrelati. Poiché la lettura di blocchi vicini riduce il tempo di seek, è importante memorizzare blocchi contenenti record interrelati il "più vicino possibile" su disco.

Gestore del buffer:
L'obiettivo principale delle strategie di memorizzazione è di minimizzare gli accessi a memoria non volatile. Il modo per minimizzare gli accessi a disco consiste nel mantenere più blocchi possibile in memoria principale in modo da evitare riletture da memoria non volatile.
La lettura di un record richiede che la pagina corrispondente sia prima portata in memoria, in un'area gestita dal DBMS detta buffer. La gestione del buffer è fondamentale dal punto di vista prestazionale ed è demandata al gestore del buffer (Buffer Manager, BR). Il buffer è organizzato in pagine, che hanno la stessa dimensione delle pagine/blocchi su disco.
A fronte di una richiesta di una pagina, ad esempio durante un'operazione di lettura (esecuzione di una interrogazione) il Buffer Manager (BM) opera come segue:

- Se la pagina è già nel buffer, viene fornito al programma chiamante l'indirizzo della pagina corrispondente;
- Se la pagina non è in memoria, il BM seleziona una pagina nel buffer per la pagina richiesta, e se la pagina prescelta dal buffer è occupata, questa viene riscritta su disco solo se è stata modificata e non ancora salvata su disco e se nessuno la sta usando. A questo punto il BM può leggere la pagina da disco e copiarla nella pagina del buffer selezionata, rimpiazzando così quella prima presente.

Quando una pagina disco è presente nel buffer, il DBMS può effettuare le sue operazioni di lettura e scrittura direttamente su di essa. Importante: Diversi ordini di grandezza di differenza tra i tempi di accesso a disco e i tempi di accesso al buffer; accedere alle pagine nel buffer invece che alle corrispondenti pagine su disco influenza notevolmente le prestazioni.

Organizzazione dei record nei file e strutture ausiliarie di accesso

A seconda dell'organizzazione primaria dei dati scelta il file che contiene i record dei dati può essere:

- file heap (o file pila): i record dati vengono memorizzati uno dopo l'altro in ordine di inserimento;
- file ordinato (o file sequenziale o file clusterizzato): i record dati sono memorizzati mantenendo l'ordinamento su uno o più campi;
- file hash: i record che condividono lo stesso valore per uno o più campo sono memorizzati «vicini» (in genere nello stesso bucket).

Le organizzazioni ordinate e hash si possono ottenere usando opportune strutture ausiliarie di accesso.

File heap

colloc	dataNol	codCli	dataRest
1111	01-Mar-2006	6635	02-Mar-2006
1115	01-Mar-2006	6635	02-Mar-2006
1118	10-Mar-2006	6642	11-Mar-2006
1117	02-Mar-2006	6635	06-Mar-2006
1118	02-Mar-2006	6635	06-Mar-2006
1111	04-Mar-2006	6642	05-Mar-2006
1119	08-Mar-2006	6635	10-Mar-2006
1120	08-Mar-2006	6635	10-Mar-2006
1116	08-Mar-2006	6642	09-Mar-2006
1121	15-Mar-2006	6635	18-Mar-2006
1122	15-Mar-2006	6635	18-Mar-2006
1113	15-Mar-2006	6635	18-Mar-2006
1129	15-Mar-2006	6635	20-Mar-2006
1119	15-Mar-2006	6642	16-Mar-2006
1126	15-Mar-2006	6610	16-Mar-2006
1112	16-Mar-2006	6610	18-Mar-2006
1114	16-Mar-2006	6610	17-Mar-2006
1128	18-Mar-2006	6642	20-Mar-2006
1124	20-Mar-2006	6610	21-Mar-2006
1115	20-Mar-2006	6610	21-Mar-2006
1124	21-Mar-2006	6642	22-Mar-2006
1117	21-Mar-2006	6610	?
1127	22-Mar-2006	6635	?
1125	22-Mar-2006	6635	?
1122	22-Mar-2006	6642	?
1113	22-Mar-2006	6642	?
1116	21-Mar-2006	6610	?

File ordinato su dataNol

colloc	dataNol	codCli	dataRest
1111	01-Mar-2006	6635	02-Mar-2006
1115	01-Mar-2006	6635	02-Mar-2006
1117	02-Mar-2006	6635	06-Mar-2006
1118	02-Mar-2006	6635	06-Mar-2006
1111	04-Mar-2006	6642	05-Mar-2006
1119	08-Mar-2006	6635	10-Mar-2006
1120	08-Mar-2006	6635	10-Mar-2006
1116	08-Mar-2006	6642	09-Mar-2006
1118	10-Mar-2006	6642	11-Mar-2006
1121	15-Mar-2006	6635	18-Mar-2006
1122	15-Mar-2006	6635	18-Mar-2006
1113	15-Mar-2006	6635	18-Mar-2006
1129	15-Mar-2006	6635	20-Mar-2006
1119	15-Mar-2006	6642	16-Mar-2006
1126	15-Mar-2006	6610	16-Mar-2006
1112	16-Mar-2006	6610	18-Mar-2006
1114	16-Mar-2006	6610	17-Mar-2006
1128	18-Mar-2006	6642	20-Mar-2006
1124	20-Mar-2006	6610	21-Mar-2006
1115	20-Mar-2006	6610	21-Mar-2006
1124	21-Mar-2006	6642	22-Mar-2006
1116	21-Mar-2006	6610	?
1117	21-Mar-2006	6610	?
1127	22-Mar-2006	6635	?
1125	22-Mar-2006	6635	?
1122	22-Mar-2006	6642	?
1113	22-Mar-2006	6642	?

File hash su dataNol

colloc	dataNol	codCli	dataRest
1117	02-Mar-2006	6635	06-Mar-2006
1118	02-Mar-2006	6635	06-Mar-2006
1111	04-Mar-2006	6642	05-Mar-2006
1119	08-Mar-2006	6635	10-Mar-2006
1120	08-Mar-2006	6635	10-Mar-2006
1116	08-Mar-2006	6642	09-Mar-2006
1111	01-Mar-2006	6635	02-Mar-2006
1115	01-Mar-2006	6635	02-Mar-2006
1118	10-Mar-2006	6642	11-Mar-2006
1121	15-Mar-2006	6635	18-Mar-2006
1122	15-Mar-2006	6635	18-Mar-2006
1113	15-Mar-2006	6635	18-Mar-2006
1129	15-Mar-2006	6635	20-Mar-2006
1119	15-Mar-2006	6642	16-Mar-2006
1126	15-Mar-2006	6610	16-Mar-2006
1112	16-Mar-2006	6610	18-Mar-2006
1114	16-Mar-2006	6610	17-Mar-2006
1124	20-Mar-2006	6610	21-Mar-2006
1115	20-Mar-2006	6610	21-Mar-2006
1128	18-Mar-2006	6642	20-Mar-2006
1124	21-Mar-2006	6642	22-Mar-2006
1116	21-Mar-2006	6610	?
1117	21-Mar-2006	6610	?
1127	22-Mar-2006	6635	?
1125	22-Mar-2006	6635	?
1122	22-Mar-2006	6642	?
1113	22-Mar-2006	6642	?

Esempio 1: *restituire tutti i noleggi*

- L'ordine di memorizzazione dei record nel file è ininfluente a fini prestazionali perché devono essere restituite tutte le tuple;
- L'organizzazione a heap va benissimo.

Esempio 2: *Restituire tutti i noleggi effettuati in data '15−mar−2006'*

- Se i record sono organizzati a heap nel file, devo comunque accedere tutto il file per recuperare i record che soddisfano la richiesta;
- Tanti blocchi potrebbero essere acceduti anche se non contengono alcun record da restituire come risultato (Esempio: File composto da 10.000 blocchi e 1 solo noleggio effettuato in data 15-mar-2006)

Esempio 3: *Restituire tutti i noleggi effettuati in data '15−mar−2006'*

- Se i record fossero memorizzati ordinati nel file, si potrebbe procedere sequenzialmente, fino al primo record con dataNol = '15-mar-2006' e leggere fino al primo record con dataNol > '15-mar-2006';
- Si riduce il numero di blocchi letti;
- Si possono comunque leggere blocchi che non contengono risultati;
- Tutti i blocchi contenenti i film con dataNol < '15-mar-2006' non contengono dati utili per la richiesta (precedono i blocchi contenenti i record di nostro interesse se il file è ordinato rispetto a dataNol);
- Se si potesse 'ricordare' la posizione del primo blocco contenente record con dataNol = '15-mar-2006', si ridurrebbe il numero di blocchi 'inutili' letti.

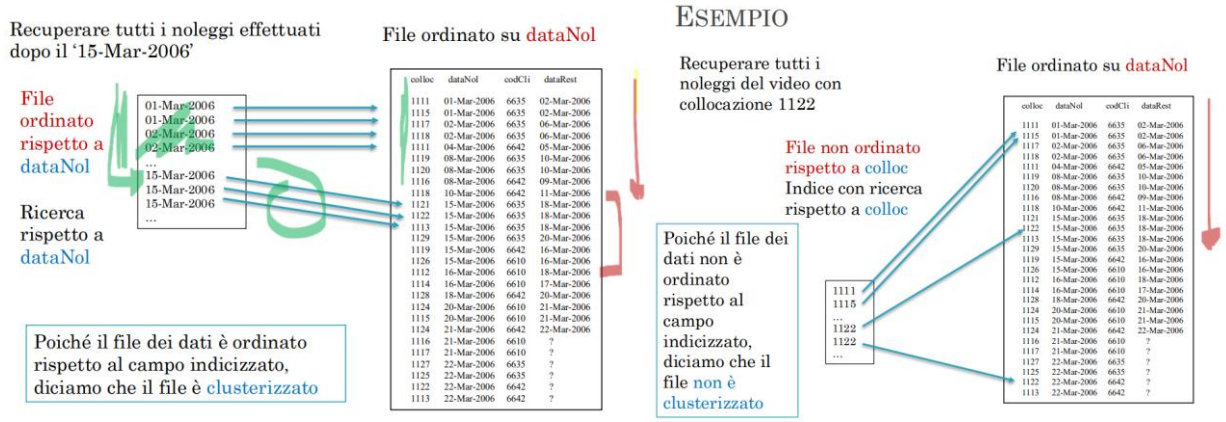
Osservazioni:

1. Le organizzazioni dei record nei file (heap, sequenziale, hash) permettono di migliorare le prestazioni di ricerche rispetto a condizioni su campi/attributi che sono quelli utilizzati per organizzare i record nel file;
2. Possono però portare ad accedere blocchi inutili;
3. Inoltre non sono efficaci per ricerche con condizioni su altri attributi rispetto a quelli considerati per organizzare i record nel file.

Indici:

Per ovviare a questi limiti si creano delle strutture ausiliarie di accesso, chiamate anche indici, che forniscono cammini di accesso alternativi ai dati, per localizzare velocemente i dati di interesse. Questi cammini di accesso permettono di determinare direttamente i record che verificano una data query. Le strutture ausiliarie di accesso permettono di eseguire in maniera più efficiente operazioni di ricerca (spesso basate su condizioni di selezione e join), rispetto a uno o più campi del record, chiamati chiave di ricerca. Chiave di ricerca, chiave primaria, chiave esterna sono concetti distinti:

- La chiave di ricerca si riferisce sempre a un indice (quindi a una struttura del livello fisico della base di dati);
- Le chiavi primarie o esterne sono vincoli di integrità che si riferiscono al livello logico della base di dati.



Operazioni di ricerca (selezione)

Gli indici permettono di velocizzare l'esecuzione di operazioni di selezione (e join). Consideriamo diversi tipi di condizioni:

- uguaglianza su chiave di ricerca primaria (il video con collocazione 1001);
- uguaglianza su chiave di ricerca secondaria (i film di genere drammatico);
- range (i film con valutazione tra 2 e 5);
- combinazioni delle precedenti (i film di genere drammatico con valutazione tra 2 e 5).

Chiave di ricerca

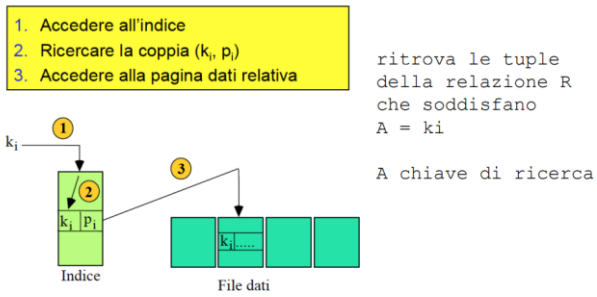
Per ovviare a questi limiti si creano delle strutture ausiliarie di accesso, chiamate anche indici, che forniscono cammini di accesso alternativi ai dati, per localizzare velocemente i dati di interesse. Questi cammini di accesso permettono di determinare direttamente i record che verificano una data query. Le strutture ausiliarie di accesso permettono di eseguire in maniera più efficiente operazioni di ricerca (spesso basate su condizioni di selezione e join), rispetto ad una certa chiave di ricerca. Si usa il termine chiave di ricerca per indicare un attributo o insieme di attributi usati per la ricerca (anche diversi dalla chiave primaria).

Un indice su una relazione R è un insieme di coppie del tipo (k_i, r_i) dove:

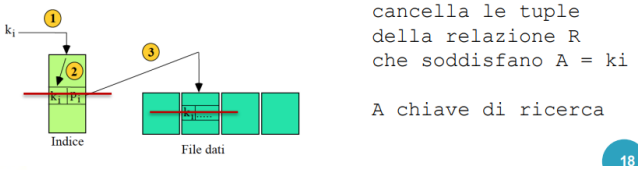
- k_i è un valore per la chiave di ricerca su cui l'indice è costruito;
- r_i può assumere diverse forme, per il momento assumiamo che sia un puntatore a un record con valore di chiave k_i; è quindi un RID o, al limite, un PID.

L'indice contiene una coppia per ogni tupla di R. Possono esistere più coppie con lo stesso valore k_i ed esiste esattamente una coppia per un certo valore r_i. Il vantaggio di usare un indice nasce dal fatto che la chiave è solo parte dell'informazione contenuta in un record. Pertanto, l'indice occupa uno spazio minore rispetto al file dati.

ACCESSO CON INDICE: SCHEMA GENERALE



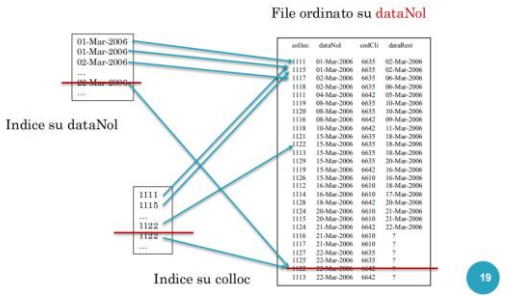
L'uso di indici rende l'esecuzione delle interrogazioni più efficiente, ma rende più costosi gli aggiornamenti. Un'interrogazione non è mai rallentata dalla presenza di indici, al più non vengono utilizzati per la sua esecuzione. A seguito di un aggiornamento dei dati tutti gli indici devono essere aggiornati.



18

ESEMPIO

```
cancella le tuple
della relazione Noleggio
che soddisfano
dataNol = '22-Mar-2006' AND colloc = 1122
```

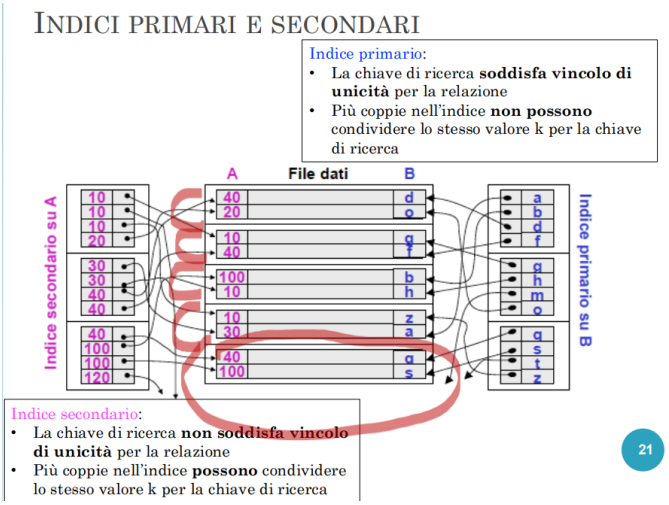


19

Se la chiave di ricerca di un indice I è anche chiave per la relazione, quante coppie in I possono condividere lo stesso valore k per la chiave di ricerca? Una sola coppia; in questo caso l'indice si chiama primario.

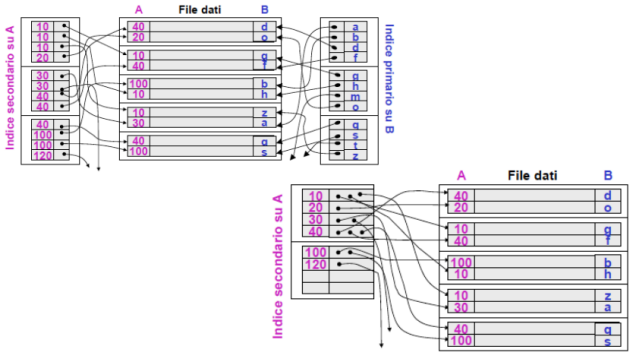
Se la chiave di ricerca di un indice I non è chiave per la relazione, quante coppie in I possono condividere lo stesso valore k per la chiave di ricerca? Anche più di una; in questo caso l'indice si chiama secondario.

Esempio: per Video, indice su colloc è indice primario, mentre indice su titolo è indice secondario.



Indici secondari a liste di puntatori

Per evitare di replicare inutilmente i valori di chiave, la soluzione più comunemente adottata per gli indici secondari consiste nel raggruppare tutte le coppie con lo stesso valore di chiave in una lista di puntatori.



Tipi di indici

Le diverse tecniche di indice differiscono nel modo in cui organizzano l'insieme di coppie (k_i, r_i):

- Indici ordinati (anche chiamati indici ad albero): le coppie (k_i, r_i) vengono mantenute esplicitamente. Le coppie vengono salvate in un file, ordinate rispetto ai valori di chiave k_i.
- Indici hash: esiste una funzione hash h per cui, se una coppia (k_i, r_i) appartiene all'indice allora $h(k_i) = r_i$. Le coppie non vengono memorizzate su un file apposito ma calcolate attraverso l'uso di una funzione hash.

Indici ordinati / ad albero

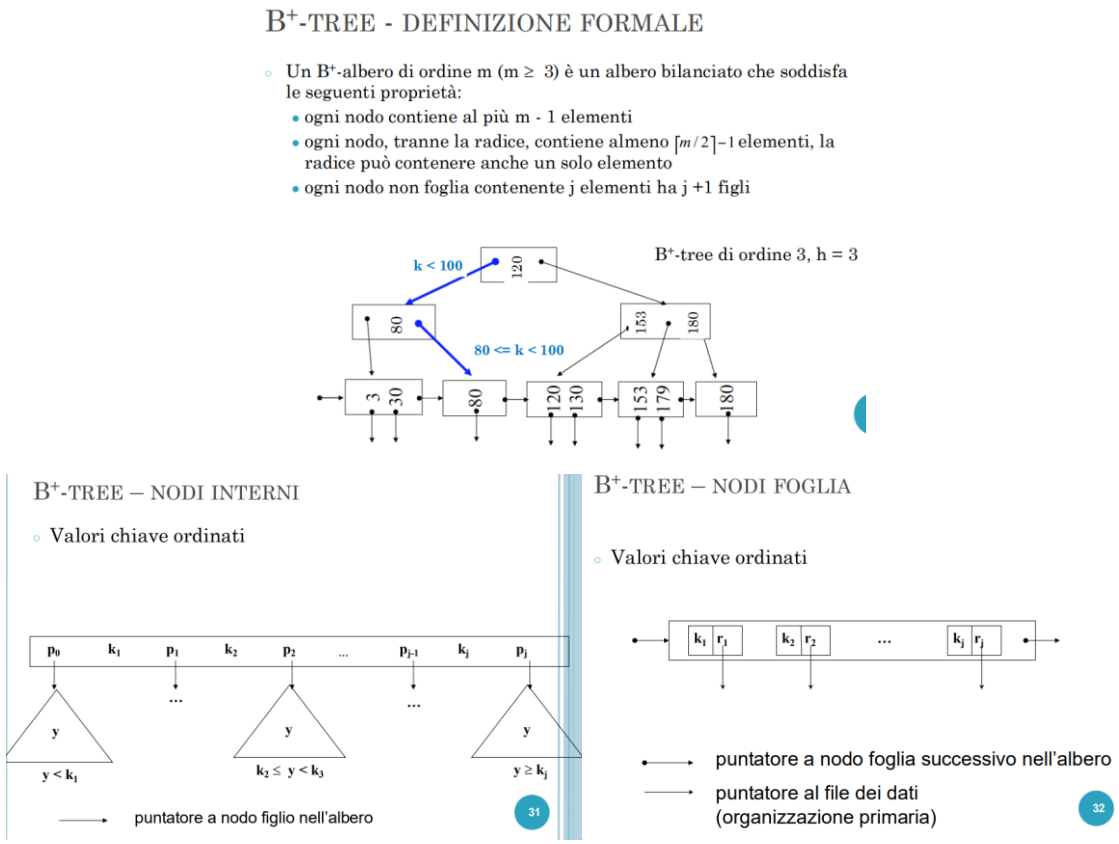
Gli indici ordinati sono indici in cui l'insieme delle coppie (k_i, r_i) vengono mantenute memorizzate in un file su disco, ordinate rispetto ai valori della chiave di ricerca k_i. I file che memorizzano gli indici costituiscono l'organizzazione secondaria dei dati.
Se l'indice è piccolo, può essere tenuto in memoria principale; molto spesso, però, è necessario tenerlo su disco e la scansione dell'indice può richiedere parecchi trasferimenti di blocchi. Viene quindi utilizzata una struttura multilivello che permette di accedere più velocemente alle coppie dell'indice; l'indice assume una struttura ad albero in cui ogni nodo dell'albero corrisponde a un blocco disco.

Gli indici ad albero sono organizzazioni ad albero bilanciato, utilizzate come strutture di indicizzazione per dati in memoria secondaria. I requisiti fondamentali per indici per memoria secondaria sono:

- bilanciamento: l'indice deve essere bilanciato rispetto ai blocchi e non ai singoli nodi (è il numero di blocchi acceduti a determinare il costo I/O di una ricerca);
- occupazione minima: è importante che si possa stabilire un limite inferiore all'utilizzazione dei blocchi, per evitare un sotto-utilizzo della memoria;
- efficienza di aggiornamento: le operazioni di aggiornamento devono avere costo limitato.

B^+-tree: caratteristiche generali

Esistono diverse strutture ad albero, la struttura più utilizzata dai DBMS commerciali è il B^+-tree, dove ogni nodo corrisponde ad un blocco. Le operazioni di ricerca, inserimento e cancellazione hanno costo, nel caso peggiore lineare nell'altezza dell'albero h e logaritmico nel numero di valori distinti N della chiave di ricerca, in quanto si può provare che h è in O(log N).
Il numero massimo m-1 di elementi memorizzabili in un nodo è l'unico parametro dipendente dalle caratteristiche della memoria, cioè dalla dimensione del blocco. Garantiscono un'occupazione di memoria almeno del 50% (almeno metà di ogni blocco allocato è effettivamente occupato).
Le coppie (k_i, r_i) sono tutte contenute in nodi (quindi blocchi) foglia e le foglie sono collegate a lista mediante puntatori (PID) per favorire ricerche di tipo range. La ricerca di una chiave deve individuare una foglia. I nodi interni memorizzano dei separatori (anch'essi valori della chiave di ricerca k_i) la cui funzione è determinare il giusto cammino nella ricerca di una chiave (parziale duplicazione delle chiavi (perché alcune sono presenti in nodi foglia e nodi interni)).



Il sottoalbero sinistro di un separatore contiene valori di chiave minori del separatore, quello destro valori di chiave maggiori od uguali al separatore. Nel caso di chiavi alfanumeriche facendo uso di separatori di lunghezza ridotta, tramite tecniche di compressione, si risparmia spazio.

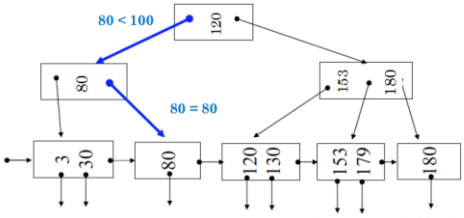
B^+-tree: ricerca di un elemento

Si trasferisce la radice in memoria e si esegue la ricerca tra le chiavi contenute per determinare se scendere nel sottoalbero sinistro o destro. Una volta raggiunta una foglia, o la chiave cercata è presente in tale foglia o non è presente nell'albero. Il costo della ricerca di una chiave nell'indice è il numero di nodi letti, sempre pari all'altezza dell'albero h.

Due casi: Ricerca per uguaglianza e Ricerca per intervallo (range).

B+-TREE – RICERCA PER UGUAGLIANZA

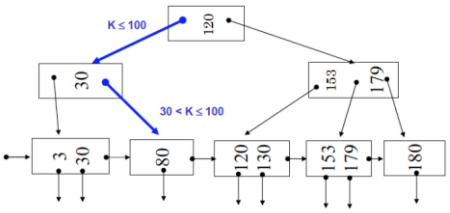
- R(K,B,C)
- Chiave di ricerca per indice primario: K
- Ricerca tuple con K = 80



Trovato, si naviga il puntatore al file dei dati e si recupera **la tupla** **(una sola, indice primario, un solo puntatore)**

B+-TREE – RICERCA PER UGUAGLIANZA

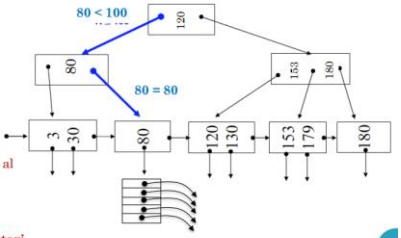
- R(K,B,C)
- Chiave di ricerca per indice primario: K
- Ricerca tuple con K = 95



Non trovato: il valore non esiste nell'indice, quindi non esistono tuple che soddisfano la condizione non si accede il file dei dati

B+-TREE –RICERCA PER UGUAGLIANZA

- R(K,B,C)
- Chiave di ricerca per indice secondario: K
- Ricerca tuple con K = 80

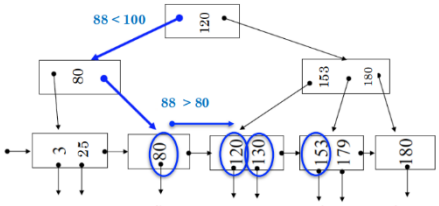


Trovato, si naviga il puntatore al file dei dati e si recuperano **le tuple** **(anche più di una, indice secondario, numero arbitrario di puntatori, inseriti in un blocco puntato dal puntatore nell'indice)**

Ricerca per intervallo: Si esegue la ricerca dell'estremo sinistro dell'intervallo. Una volta raggiunta una foglia, ci si sposta nella lista dei nodi foglia fino a superare l'estremo destro dell'intervallo. La ricerca per intervallo è quindi molto efficiente.

B+-TREE – RICERCA PER INTERVALLO

- R(K,B,C)
- Chiave di ricerca per indice primario: K
- Ricerca tuple con 88 <= K <= 154



Si navigano i puntatori di tutti i valori che soddisfano la condizione

B^+-tree: inserimento e cancellazione

Inserimento e cancellazione richiedono prima di tutto un'operazione di ricerca; possono poi richiedere ulteriori aggiustamenti dell'albero per mantenere la proprietà di bilanciamento ed i vincoli sul numero minimo e massimo di elementi. L'idea chiave di inserimento e cancellazione: le modifiche partono sempre dalle foglie e l'albero cresce o si accorcia verso l'alto.

Anche il costo di inserimento e cancellazione è lineare in h, cioè nell'altezza dell'albero, quindi nel logaritmo del numero di valori memorizzati.

Inserimento:

Supponiamo di inserire una nuova tupla t_i in una relazione R(K,B,C). Per prima cosa, il record tr_i corrispondente a t_i viene aggiunto nel file corrispondente a R. Supponiamo che esista un indice su R(B). La coppia da inserire nell'indice è (k_i, r_i), con $k_i = t_i$. B e r_i identificatore di tr_i.

La procedura di inserimento procede innanzitutto cercando k_i nell'indice. Se la foglia a cui si arriva contiene k_i, si aggiunge semplicemente r_i alla lista dei puntatori. Se la foglia a cui si arriva non contiene k_i. Se la foglia non è piena, si inserisce la nuova coppia (k_i, r_i) e si riscrive la foglia così aggiornata. Se la foglia è piena, si attiva un processo di suddivisione della foglia (quindi si crea una nuova foglia) che può propagarsi al livello superiore e, nel caso peggiore, fino alla radice; i nodi ai livelli superiori non sono necessariamente pieni e quindi possono "assorbire" le informazioni che si propagano dalle foglie, e la propagazione degli effetti sino alla radice può provocare l'aumento dell'altezza dell'albero.

Cancellazione:

Supponiamo di cancellare una tupla t_i da una relazione R(K,B,C). Per prima cosa, il record tr_i corrispondente a t_i viene individuato nel file corrispondente a R. Supponiamo che esista un indice su R(B). La coppia da cancellare nell'indice è (k_i, r_i), con $k_i = t_i$. B e r_i identificatore di tr_i. Viene innanzitutto effettuata la ricerca di k_i, determinando così una foglia. Supponiamo di trovare la foglia che contiene k_i (altrimenti non c'è nulla da cancellare): se la foglia non è troppo vuota (cioè ha ancora il numero minimo di elementi dopo la cancellazione), si cancella la chiave e si riscrive la foglia così aggiornata; se la foglia è troppo vuota, si attiva un processo di concatenazione o un processo di bilanciamento con foglie adiacenti.

Procedimento inverso rispetto a inserimento.

B^+-tree: altezza

- altezza = numero di nodi che compaiono in un cammino dalla radice ad un nodo foglia;
- i B^+-tree permettono di prevedere con sufficiente approssimazione l'altezza media dell'albero in funzione delle chiavi presenti;
- si possono stimare i costi della ricerca e delle operazioni di aggiornamento.

◦ Altezza di un B+-albero in funzione del numero dei nodi e dell'ordine, supponendo che le chiavi siano di 10 byte ed i puntatori di 4 byte

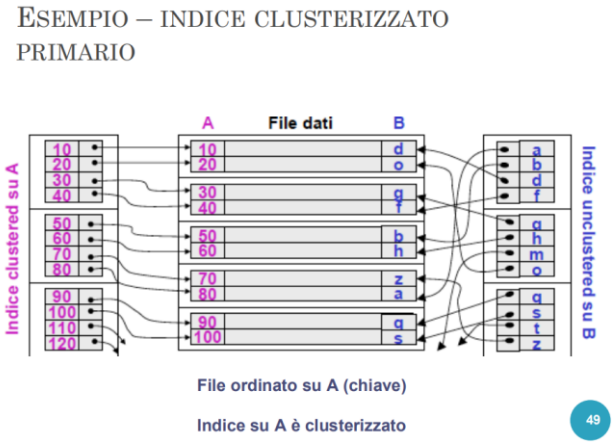
dim pagina	m	$N = 1000$		$N = 10000$		$N = 100000$		$N = 1000000$	
		h_{min}	h_{max}	h_{min}	h_{max}	h_{min}	h_{max}	h_{min}	h_{max}
512	36	1,9	3,2	2,6	3,9	3,2	4,7	3,9	5,5
1024	73	1,6	2,7	2,1	3,4	2,7	4,0	3,2	4,6
2048	146	1,4	2,4	1,8	3,0	2,3	3,5	2,8	4,1
4096	292	1,2	2,2	1,6	2,7	2,0	3,2	2,4	3,6

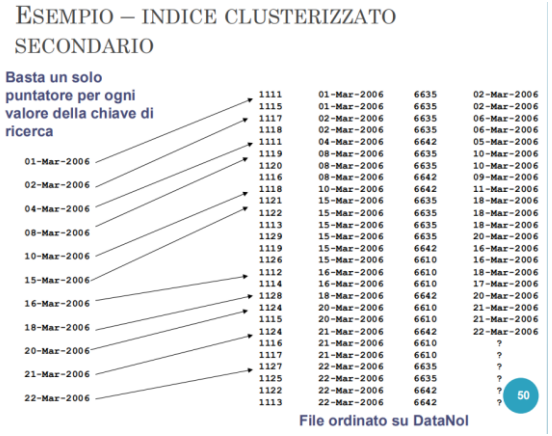
B^+-tree: varianti

B-tree: Variante in cui le entrate dell'indice sono associate alle chiavi anche nei nodi interni dell'albero, senza duplicazione delle chiavi. Migliora l'occupazione di memoria; la ricerca di una singola chiave è più costosa in media in un B^+-tree (si deve necessariamente raggiungere sempre la foglia per ottenere il puntatore ai dati). Un B^+-tree è meglio per ricerche di intervalli di valori.

Indici ad albero clusterizzati e non clusterizzati:

Un indice ad albero è clusterizzato se il file dei dati è ordinato rispetto alla chiave di ricerca, in caso contrario si definisce non clusterizzato. Un file dei dati ordinato è sempre associato a un indice ad albero clusterizzato: si sfrutta ordinamento delle foglie dell'indice per ordinare il file dei dati. Le operazioni di inserimento, cancellazione e aggiornamento nel file ordinato sono facilitate dalla presenza dell'indice. Al più un indice clusterizzato per tabella (indipendentemente dal tipo).





INDICI AD ALBERO SU SINGOLO ATTRIBUTO E MULTI-ATTRIBUTO

Indici ad albero su singolo attributo e multi-attributo:

- Indice su singolo attributo: indice la cui chiave di ricerca è costituita da un solo attributo;
- Indice multiattributo:indice la cui chiave di ricerca è costituita da più di un attributo.

Esempio per Noleggio: indice su (dataNol,colloc) è indice multiattributo; indice su codCli è indice a singolo attributo.

Un indice multiattributo è definito su una lista di attributi; diversi ordinamenti degli attributi chiave corrispondono a diversi indici. Indice su (dataNol, colloc) è diverso da indice su (colloc, dataNol).

Un indice multi-attributo su A1, A2, …, An permette di determinare direttamente:

- le tuple che soddisfano condizioni di uguaglianza su tutti gli attributi A1,…, An o sui primi i < n;
- le tuple che soddisfano condizioni di uguaglianza sui primi i attributi A1,…, Ai e condizioni di intervallo sull'attributo Ai+1.

Un indice multiattributo può supportare più interrogazioni di un indice a singolo attributo; deve però essere aggiornato più di frequente ed è di dimensione maggiore rispetto ad indice su singolo attributo.

Indici hash

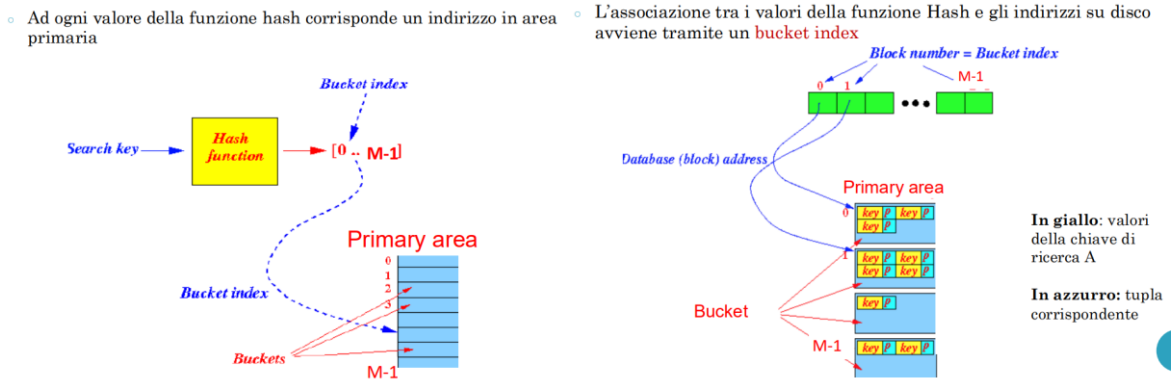
L'uso di indici ad albero ha lo svantaggio di richiedere la scansione di una struttura dati, memorizzata su disco, per localizzare i dati. Questo perché le associazioni (k_i, r_i) vengono mantenute in forma esplicita, come record in un file. Gli indici hash al contrario mantengono le associazioni (k_i, r_i) in modo implicito, tramite l'uso di una funzione hash, definita sul dominio della chiave di ricerca.

Caratteristiche generali

Data una chiave di ricerca A per una relazione R, con dominio D_A, una funzione hash per A è una funzione surgettiva: $H(D_A) \rightarrow \{0,\ldots,M\text{-}1\}$.

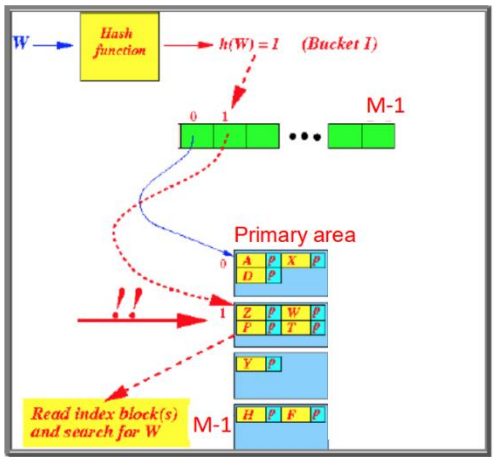
Assumiamo M costante. Ad ogni valore v per la chiave di ricerca, il valore restituito dalla funzione hash corrisponde a un indirizzo di una pagina logica, o bucket, che può corrispondere a uno o più blocchi su disco e che contiene i record con valore v per A. Per semplicità iniziare a pensare che un bucket corrisponda a un singolo blocco. La corrispondenza tra valori generati dalla funzione e

indirizzi su disco viene mantenuta in una bucket directory (o bucket index). L'area di memoria su disco su cui vengono memorizzati i bucket viene chiamata area primaria. Quindi, un indice hash contiene tutte le coppie $(k_i, H(k_i))$, con k_i in D_A.



Ricerca per uguaglianza

1. Supponiamo di cercare le tuple della relazione R(K,B,C) con B = w, con indice hash su attributo B;
2. Si calcola H(w);
3. Se, ad esempio, H(w) = 1, tramite il bucket index, si determina l'indirizzo corrispondente al bucket (supponiamo 1);
4. Si accede il bucket e, record per record, si cercano le tuple t con t.B = w.



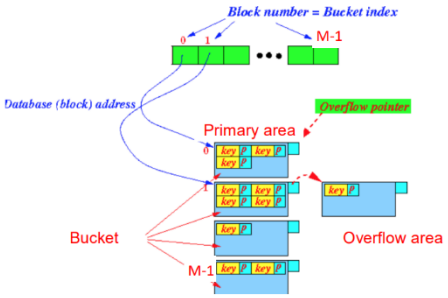
Ricerca per intervallo

Un indice hash non supporta ricerche per intervallo. La funzione hash, infatti, non mantiene l'ordine: tuple con valori contigui per la chiave di ricerca possono essere memorizzati in bucket diversi.

Inserimento e trabocchi

Per inserire un record, si procede come per la ricerca per uguaglianza, identificando il bucket in cui il record deve essere inserito. Il numero di record che possono essere allocati nello stesso bucket determina la capacità c dei bucket. I bucket hanno in genere la stessa capacità. Se il bucket individuato per l'inserimento ha ancora spazio (la sua capacità è inferiore a c), si inserisce il record nel bucket. Se il bucket non ha più spazio (la sua capacità è pari a c), si genera un trabocco (overflow). La presenza di overflow può richiedere l'uso di un'area di memoria separata, detta area di overflow.

- Supponiamo che ogni bucket abbia capacità c = 4 (può contenere 4 record);
- Supponiamo adesso di inserire un nuovo record con chiave k tale che H(k) = 1;
- Il bucket 1 è già pieno, quindi è necessario usare un blocco in area di overflow per memorizzare il nuovo record.



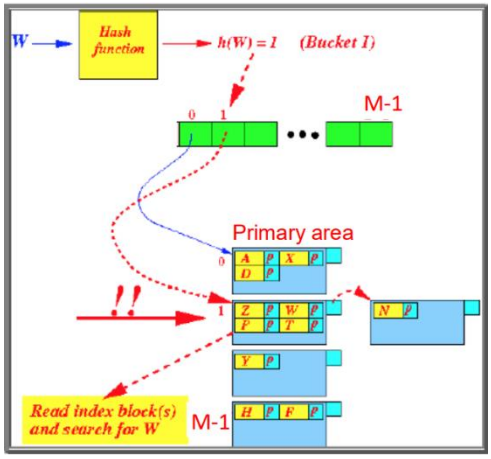
Area primaria e area di overflow

L'area primaria viene allocata al momento della creazione dell'indice hash, l'area di overflow successivamente, quando ce n'è bisogno. Il sistema ottimizza l'allocazione dell'area primaria: se un bucket contiene più blocchi in area primaria, questi sono memorizzati «vicini» su memoria secondaria

(nello stesso cilindro su hard disk). Tempo di latenza minimizzato per accesso ad area primaria. Il sistema non è in grado di ottimizzare l'allocazione dell'area di overflow: i blocchi dell'area di overflow vengono memorizzati dove si può; in generale, maggiore tempo di latenza per accesso ad area di overflow.

Ricerca per uguaglianza con trabocchi

1. Supponiamo di cercare le tuple della relazione R(K,B,C) con B = w, con indice hash su attributo B;
2. Si calcola H(w);
3. Se, ad esempio H(w) = 1, tramite il bucket index, si determina l'indirizzo corrispondente al bucket (supponiamo 1);
4. Si accede il bucket e, record per record, si cercano le tuple t con t.B = w;
5. Si deve analizzare sia la parte del bucket in area primaria sia quella in area di overflow.



Cancellazione

- Si cercano i record da cancellare, come descritto per l'operazione di ricerca per uguaglianza;
- Si elimina il record dal bucket;
- La cancellazione di un record da un'area di overflow può determinare la cancellazione dell'area di overflow (se è l'ultimo record nell'area).

Funzione hash

Una funzione hash è un'applicazione surgettiva H dall'insieme delle possibili chiavi all'insieme 0,…,M-1 dei possibili indirizzi che verifichi le seguenti proprietà:

- distribuzione uniforme delle chiavi nello spazio degli indirizzi (ogni indirizzo deve essere generato con la stessa probabilità);
- distribuzione casuale delle chiavi (eventuali correlazioni tra i valori delle chiavi non devono tradursi in correlazioni tra gli indirizzi generati).

Tali proprietà dipendono dall'insieme delle chiavi su cui si opera e quindi non esiste una funzione universale ottima. Una funzione hash è detta perfetta se per un certo numero di record non produce trabocchi; una funzione perfetta può sempre essere definita disponendo di un'area primaria con capacità complessiva pari al numero dei record da memorizzare.

In genere le funzioni hash operano su insiemi di chiavi intere; se i valori delle chiavi sono stringhe alfanumeriche, si può associare in modo univoco ad ogni chiave un numero intero, prima di applicare la trasformazione.

Metodo della divisione

La chiave numerica viene divisa per M e l'indirizzo è ottenuto considerando il resto: H(k)=k mod M, dove mod indica il resto della divisione intera. Affinché H distribuisca bene, M deve essere primo oppure non primo con nessun fattore primo minore di 20. Test sperimentali eseguiti con file con caratteristiche molto diversificate mostrano che, in generale, il metodo della divisione è il più adattabile ed è quello più utilizzato dai DBMS (incluso PostgreSQL).

Costi

Un indice hash supporta in modo efficiente ricerche per uguaglianza, inserimenti e cancellazioni. Non è necessario accedere un file come per gli indici ordinati. In assenza di overflow, il costo di accesso a indice è costante: Costo di accesso al bucket index se non è già in memoria (1); Costo di accesso al bucket (1 se corrisponde a un singolo blocco); Costo di scrittura, per inserimento e cancellazione (1). In presenza di overflow, le prestazioni non sono facilmente determinabili.

Come crearli

La creazione di un indice hash richiede di specificare:

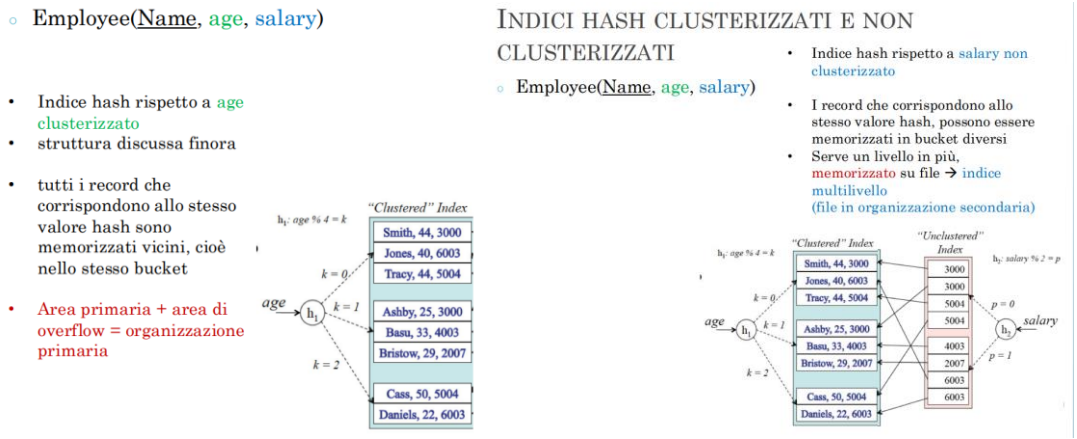
- la funzione H per la trasformazione della chiave;
- il metodo per la gestione dei trabocchi;
- il fattore di caricamento d:
    - valore tra 0 e 1 che indica quanto si intendono mantenere "pieni" i bucket;
    - influisce sulla probabilità trabocchi.

Il DBA ha al più la possibilità di agire sulla funzione, ma non in tutti i DBMS si può specificare.

Indici hash clusterizzati e non clusterizzati

Un indice hash è clusterizzato se i record che condividono lo stesso valore per la chiave di ricerca sono memorizzati in posizioni adiacenti nel file dei dati (quindi nello stesso bucket). In caso contrario, l'indice è non clusterizzato. Gli indici hash discussi finora sono clusterizzati e l'area primaria corrisponde all'organizzazione primaria dei dati. Un file dei dati di tipo hash è sempre associato a un indice hash clusterizzato: le operazioni di inserimento, cancellazione e aggiornamento nel file ordinato sono facilitate dall'uso dell'organizzazione hash. In presenza di un indice hash clusterizzato (e quindi in presenza di un file organizzato a hash) l'organizzazione primaria corrisponde ai record memorizzati nell'area primaria + i record memorizzati nell'area di overflow. Al più un indice clusterizzato per tabella (indipendentemente dal tipo).

Gli indici hash possono anche essere non clusterizzati: in questo caso l'area primaria NON corrisponde all'organizzazione primaria. L'area primaria contiene, oltre ai valori della chiave di ricerca, i puntatori all'organizzazione primaria (file dei dati). In questo caso l'area primaria dell'indice viene memorizzata su file.



Indici hash su singolo attributo e multi-attributo

Un indice hash multiattributo è definito su una lista di attributi. Diversi ordinamenti degli attributi chiave potrebbero corrispondere a diverse organizzazioni (dipende dalla funzione hash); indice su (dataNol,colloc) potrebbe essere diverso da indice su (colloc,dataNol).

Un indice multi-attributo su A1, A2, …, An permette di determinare direttamente le tuple che soddisfano condizioni di uguaglianza su tutti gli attributi A1,…, An.

CONFRONTO TRA INDICI AD ALBERO E HASH

- Per selezioni con condizioni di uguaglianza del tipo

```
Ritrova tutte le tuple
di R con Ai=C
```

un indice hash su Ai è preferibile

- Per selezioni con condizioni di tipo range del tipo

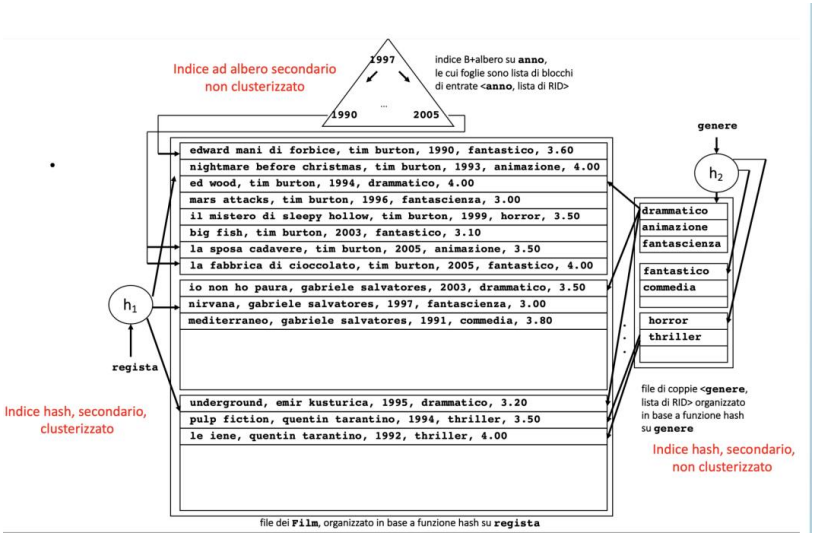
```
Ritrova tutte le tuple di R
     con C1 < Ai < C2
```

un indice ad albero su Ai è preferibile

- Gli indici hash infatti non mantengono l'ordine

- Infatti:
  - la scansione di un indice ordinato ha un costo proporzionale al logaritmo del numero di valori in R per Ai
  - in una struttura hash il tempo di ricerca è indipendente dalla dimensione della base di dati

25



2) Elaboratore di interrogazioni

Motivazioni

Query di esempio:

Consideriamo la seguente interrogazione in SQL:

SELECT B, D
FROM R, S
WHERE R.A = "c" ∧ S.E = 2 ∧ R.C=S.C

Come sappiamo SQL è dichiarativo. La stessa query si può rappresentare in algebra (linguaggio operazionale) con la seguente espressione:

ΠB,D [σR.A="c"∧ S.E=2 ∧ R.C = S.C (R X S)]

RISULTATO

R

A	B	C
a	1	10
b	1	20
c	2	10
d	2	35
e	3	45

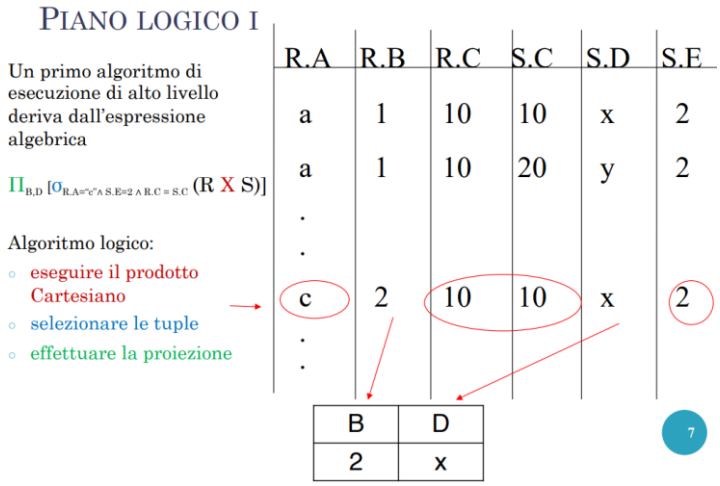
S

C	D	E
10	x	2
20	y	2
30	z	2
40	x	1
50	y	3

B	D
2	x

6

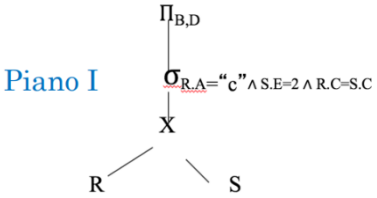
Piano logico I



L'espressione algebrica rappresenta un algoritmo (logico) di esecuzione che opera su tabelle e si può rappresentare come un albero:

ΠB,D [σR.A="c"∧ S.E=2 ∧ R.C = S.C (R X S)]

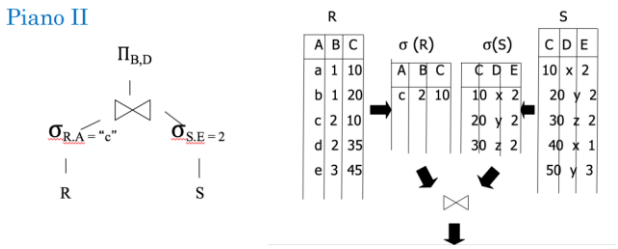
Piano di esecuzione logico: gli operatori manipolano tabelle e tuple, elementi del livello logico.



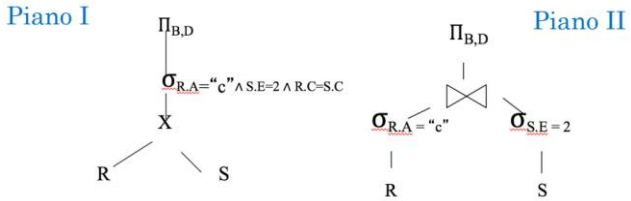
Piano logico II

Altra possibile strategia corrispondente a piano logico alternativo (quindi a espressione algebrica alternativa)

ΠB,D [σR.A="c"(R) ⋈ σS.E=2 (S)]



Piano logico I e Piano logico 2: Quale piano logico porta ad una esecuzione più efficiente?



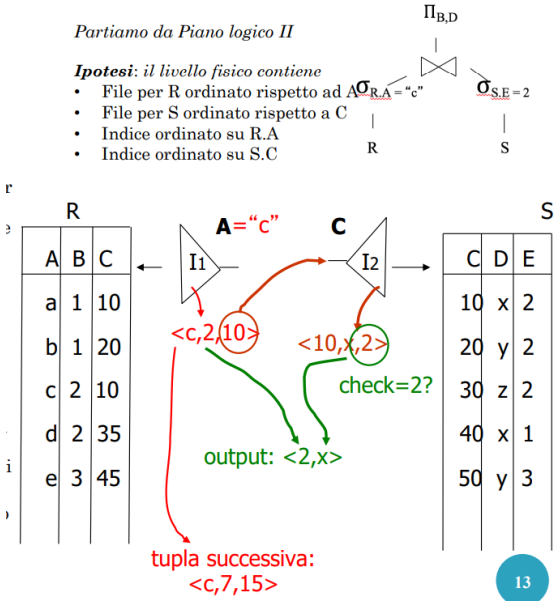
Piano II, in quanto evita l'esecuzione del prodotto cartesiano riducendo la dimensione dei risultati intermedi generati e il numero di operazioni eseguite.

Dal piano logico al piano fisico

I piani logici forniscono alcune indicazioni su come eseguire l'interrogazione ma non corrispondono ad algoritmi utilizzabili dal sistema per l'esecuzione. I dati sono memorizzati su disco quindi gli algoritmi di esecuzione delle interrogazioni devono operare su record memorizzati nei file. Dal piano di esecuzione logico al piano di esecuzione fisico, ogni operatore viene implementato tramite un algoritmo che manipola record.

Piano fisico

- Si usa l'indice su R.A per selezionare i record corrispondenti alle tuple di R con R.A = "c";
- Per ogni valore di R.C trovato, si usa l'indice su S.C per trovare i record corrispondenti alle tuple in join;
- Si eliminano i record di S tali che S.E ≠ 2;
- Si concatenano di record di R e S risultanti, proiettando su B e D e si restituisce la tupla;
- Si ripete il procedimento per ogni altra tuple restituita dall'indice su R.A.

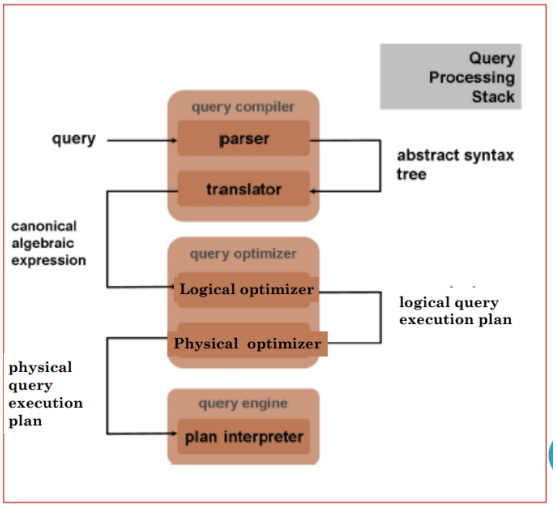


Elaborazione e ottimizzazione

Per interrogazioni complesse esistono molteplici strategie logiche possibili, molti piani di esecuzione logici. Scelto un piano logico e dato uno schema fisico, esistono molteplici algoritmi lo realizzano. Per ogni operatore algebrico esistono diversi possibili algoritmi. Algoritmi diversi possono avere costi diversi (= accedere a un numero diverso di blocchi disco). Il compito del query processor è individuare il piano fisico di esecuzione più efficiente, a partire da uno schema logico e uno schema fisico in input. Il costo di determinare la strategia ottima può essere elevato. Il vantaggio in termini di tempo di esecuzione che se ne ricava è tuttavia tale da rendere preferibile eseguire l'ottimizzazione.

Passi nell'<u>esecuzione</u> di una interrogazione

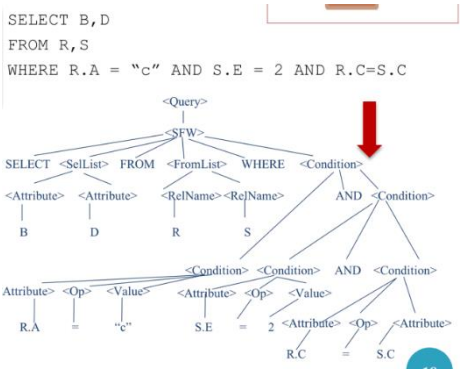
Query processor



Parser

- input: interrogazione SQL
- output: parse tree (o abstract syntax tree)

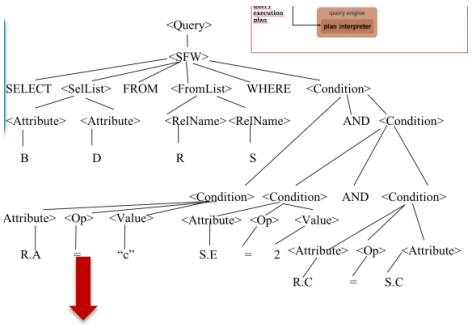
Viene controllata la correttezza sintattica della query SQL e ne viene generata una rappresentazione interna (in termini di parse tree).



Translator

- input: parse tree
- output: espressione algebrica canonica

Viene generata una espressione algebrica corrispondente a un prodotto cartesiano delle relazioni in clausola FROM seguita da condizioni di selezione, da clausola WHERE seguita da condizioni di proiezione, da clausola SELECT.

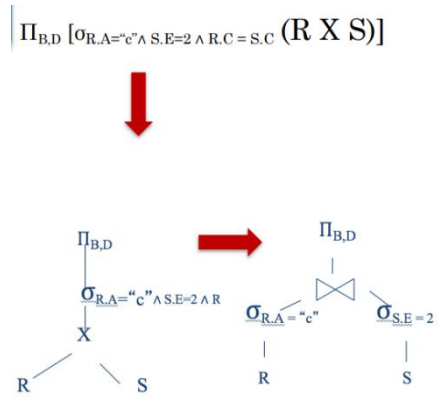


Ottimizzazione logica

- input: espressione algebrica canonica
- output: piano di esecuzione logico ottimizzato (logical query plan – LQP)

Piano di esecuzione logico: espressione algebrica (in un'algebra estesa) per l'interrogazione, rappresentata come albero. L'espressione algebrica canonica viene rappresentata come piano di esecuzione logico. Il piano di esecuzione logico iniziale viene poi trasformato in un piano equivalente ma più efficiente da eseguire, usando le proprietà dell'algebra relazionale.

$$\Pi_{B,D}\,[\sigma_{R.A=\text{“c”}\,\wedge\,S.E=2\,\wedge\,R.C\,=\,S.C}\,(R\times S)]$$



Ottimizzazione fisica

- input: piano di esecuzione logica ottimizzato
- output: piano di esecuzione fisico ottimale (physical query plan - PQP)

Piano di esecuzione fisico: algoritmo di esecuzione dell'interrogazione, rappresentato come albero, sul livello fisico. Si seleziona il piano di esecuzione fisico più efficiente, che riduce quindi gli accessi a disco. Si determina in modo preciso come la query sarà eseguita (per esempio si determina che indici si useranno). La scelta avviene considerando tutti i possibili piani fisici (= algoritmi) che realizzano il piano logico scelto, valutando il costo di ognuno di essi, usando statistiche di sistema, e scegliendo il piano fisico di minor costo.

Esecuzione del piano

- input: piano di esecuzione fisico ottimale
- output: risultato della interrogazione

Il piano di esecuzione fisico viene eseguito sul livello fisico e il risultato ottenuto viene restituito come risultato dell'interrogazione.

Ottimizzazione logica

Operatori logici

I DBMS relazionali considerano un'algebra relazionale estesa, che include operazioni algebra relazionale e altri operatori presenti in SQL: unione, intersezione, differenza senza eliminazione dei duplicati; proiezione senza eliminazione dei duplicati; ordinamento; raggruppamento.

Approccio

L'ottimizzazione logica si basa su equivalenze algebriche. Due espressioni e1 ed e2 dell'algebra relazionale (o due LQP) sono dette equivalenti se, per ogni possibile base di dati in input D, producono lo stesso risultato in output quando vengono eseguite su D.

ΠB,D [σR.A="c"∧ S.E=2 ∧ R.C = S.C (R X S)] equivalente a ΠB,D [σR.A="c"(R) ⋈ σS.E=2 (S)]. Tali equivalenze vengono utilizzate come regole di riscrittura, guidati da opportune euristiche per passare da un'espressione algebrica (quindi da un LQP) ad un'altra, ad essa equivalente ma più efficiente.

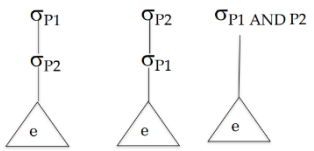
ΠB,D [σR.A="c"∧ S.E=2 ∧ R.C = S.C (R X S)] ➔ ΠB,D [σR.A="c"(R) ⋈ σS.E=2 (S)]

Equivalenze algebriche

- Selezione:

  σ P1 (σP2 (e)) ≡ σ P2 (σP1 (e)) ≡ σ P1 AND P2 (e)

  permette di gestire cascate di selezioni e stabilisce la commutatività della selezione.



- Proiezione:

  ΠA1,....,An(ΠB1,....,Bm(e)) ≡ ΠA1,....,An(e)

  permette di gestire cascate di proiezioni; vale se {A1,.....,An} ⊆ {B1,.....,Bm}.

- Commutazione di selezione e proiezione:

  se una selezione con predicato P coinvolge solo gli attributi A1,.....,An, allora

  ΠA1,....,An(σP (e)) ≡ σP (ΠA1,....,An(e))

  più in generale, se il predicato P coinvolge anche gli attributi B1,.....,Bm che non sono tra gli attributi A1,.....,An allora

  ΠA1,....,An(σP (e)) ≡ ΠA1,....,An(σP (ΠA1,....,An,B1,....,Bm(e)))

- Commutazione di selezione e prodotto Cartesiano:

  se una selezione con predicato P coinvolge solo gli attributi di e1, allora

  σP(e1 × e2) ≡ σP(e1) × e2

  come conseguenza, se P=P1 AND P2 dove P1 coinvolge solo gli attributi di e1 e P2 quelli di e2

  σP1 AND P2(e1 × e2) ≡ σP1(e1) × σP2(e2)

  inoltre se P1 coinvolge solo attributi di e1, mentre P2 coinvolge attributi di e1 e di e2

  σP1 AND P2(e1 × e2) ≡ σP2(σP1(e1) × e2)

  Sia A1,....,An una lista di attributi di cui gli attributi B1,.....,Bm siano attributi di e1, e i rimanenti C1,.....,Ck siano attributi di e2

  ΠA1,....,An(e1 × e2) ≡ ΠB1,....,Bm(e1) × ΠC1,....,Ck(e2)

- Selezioni, prodotto Cartesiano e join:

  Si può trasformare una selezione ed un prodotto cartesiano in un join, in accordo alla definizione di join

  σP (e1 × e2) ≡ e1 ⋈P e2

- Prodotto Cartesiano e join:
  - Commutatività:

    $e1 \bowtie F\ e2 \equiv e2 \bowtie F\ e1$

    $e1 \bowtie e2 \equiv e2 \bowtie e1$

    $e1 \times e2 \equiv e2 \times e1$
  - Associatività:

    $(e1 \bowtie F1\ e2) \bowtie F2\ e3 \equiv e1 \bowtie F1\ (e2 \bowtie F2\ e3)$

    $(e1 \bowtie e2) \bowtie e3 \equiv e1 \bowtie (e2 \bowtie e3)$

    $(e1 \times e2) \times e3 \equiv e1 \times (e2 \times e3)$
- Altre equivalenze (unione e differenza):
  - commutatività e associatività dell'unione:

    $(e1 \cup e2) \cup e3 \equiv e1 \cup (e2 \cup e3)$

    $e1 \cup e2 \equiv e2 \cup e1$
  - Commutazione di selezione e unione:

    $\sigma P(e1 \cup e2) \equiv \sigma P(e1) \cup \sigma P(e2)$
  - commutazione di selezione e differenza

    $\sigma P(e1 - e2) \equiv \sigma P(e1) - \sigma P(e2) \equiv \sigma P(e1) - e2$
  - commutazione di proiezione e unione

    $\Pi A1,....,An(e1 \cup e2) \equiv \Pi A1,....,An(e1) \cup \Pi A1,....An(e2)$

Euristiche

Le euristiche permettono di trasformare le equivalenze in regole di riscrittura.

Euristiche fondamentali:

1. anticipare il più possibile le operazioni che permettono di ridurre la dimensione dei risultati intermedi: selezione e proiezione
2. fattorizzare condizioni di selezione complesse e introdurre, per aumentare la possibilità di applicare regole di riscrittura

Le altre regole vengono applicate per favorire l'applicazione delle euristiche descritte sopra.

EURISTICA 1.A

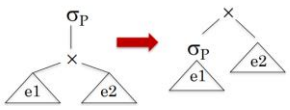
Eseguire le operazioni di selezione (σ) il più presto possibile

(una) equivalenza

$\sigma_P(e1 \times e2) \equiv \sigma_P(e1) \times e2$

(una) regola di riscrittura

$\sigma_P(e1 \times e2) \rightarrow \sigma_P(e1) \times e2$



EURISTICA 1.B

Eseguire le operazioni di proiezione (π) il più presto possibile

(alcune) equivalenze

$\Pi_{A1,...,An}(\sigma_P(e)) \equiv \sigma_P(\Pi_{A1,...,An}(e))$

$\Pi_{A1,...,An}(e1 \times e2) \equiv \Pi_{B1,...,Bm}(e1) \times \Pi_{C1,...,Ck}(e2)$

(alcune) regole di riscrittura

$\Pi_{A1,...,An}(\sigma_P(e)) \rightarrow \sigma_P(\Pi_{A1,...,An}(e))$

$\Pi_{A1,...,An}(e1 \times e2) \rightarrow \Pi_{B1,...,Bm}(e1) \times \Pi_{C1,...,Ck}(e2)$

EURISTICA 2

Fattorizzare condizioni di selezione complesse per aumentare la possibilita` di applicare regole di riscrittura

(alcune) equivalenze

$\sigma_{P1}(\sigma_{P2}(e)) \equiv \sigma_{P2}(\sigma_{P1}(e)) \equiv \sigma_{P1\ AND\ P2}(e)$

(alcune) regole di riscrittura

$\sigma_{P1}(\sigma_{P2}(e)) \leftarrow \sigma_{P2}(\sigma_{P1}(e)) \leftarrow \sigma_{P1\ AND\ P2}(e)$

Osservazione

Nessuna regola di riscrittura si riferisce all'ordine con cui eseguire un insieme di join: l'ordine di esecuzione dei join, infatti, viene deciso nella fase successiva (ottimizzazione fisica), sulla base di informazioni relative alla dimensione delle relazioni e sulla valutazione del costo dei diversi ordini di esecuzione.

Output della fase di ottimizzazione logica

L'output della fase di ottimizzazione logica è un singolo LQP ottimizzato. Un'alternativa sarebbe quella di considerare diversi equivalenti LQP nella fase di ottimizzazione fisica, ma in genere questa alternativa non viene utilizzata: per ogni piano logico restituito, bisognerebbe individuare il piano fisico ottimale, causando un aumento importante del numero di piani di cui stimare il costo.