Tecnologie della Basi Di Dati M

Antonio Davide Calì

24 aprile 2014

WWW.ANTONIOCALI.COM
Anno Accademico 2013/2014
Docenti: Marco Patella, Paolo Ciaccia

Indice

Parte I
Ottimizzazione di Interrogazioni

1 Ottimizzazione delle query

Come visto in precedenza la risoluzione di un'interrogazione (anche semplice) può essere realizzata in diversi modi e con costi differenti: basti pensare all'operatore logico join, esso può essere implementato attraverso molti algoritmi ognuno dei quali si basa sulla presenza di alcuni fattori (come ad esempio indici, o relazioni ordinatate).

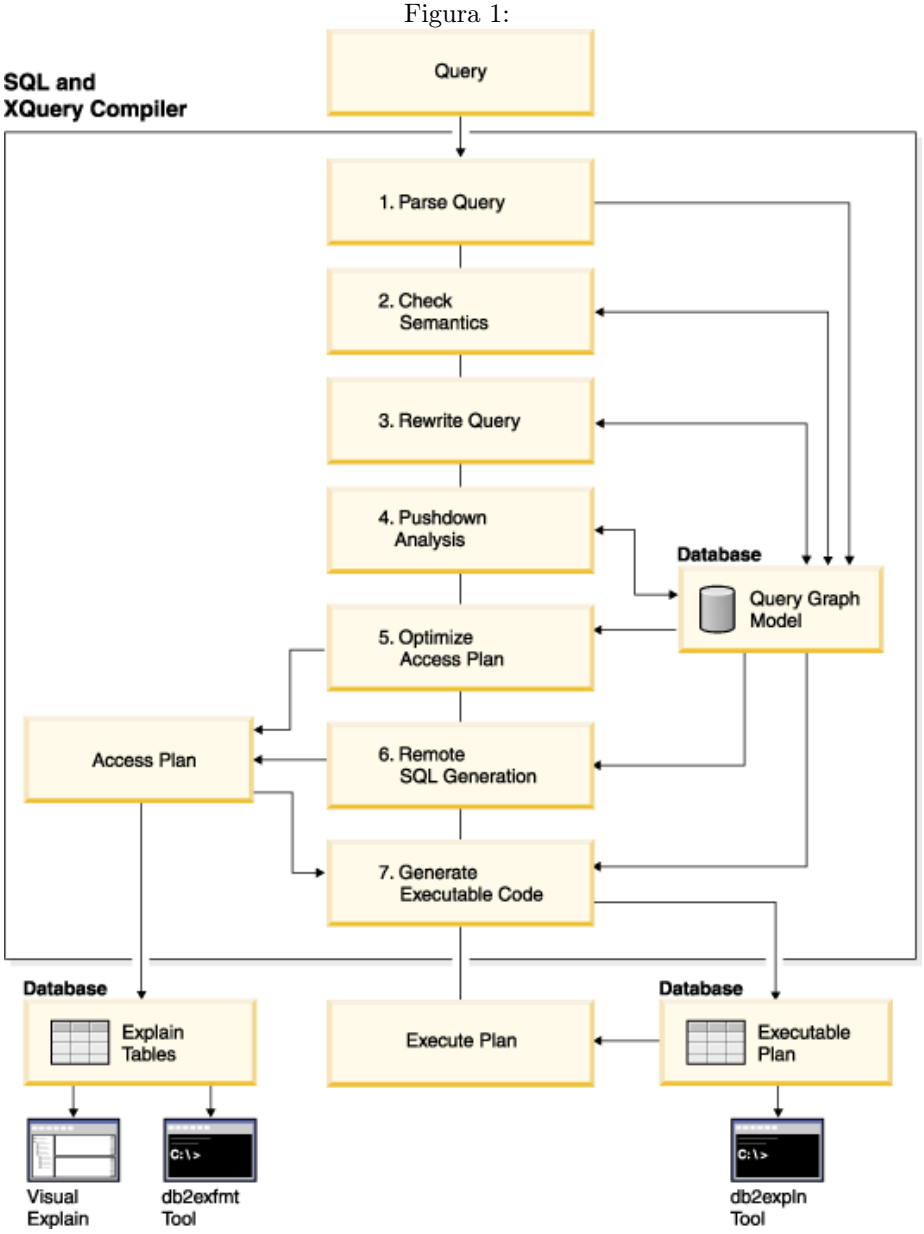
Punto fondamentale di ogni *dmbs* è riuscire a scegliere il modo più veloce per risolvere una determinata query. A tal fine il *dbms* si basa sul *query manager*, il quale dispone di un *ottimizzatore* e di un *valutatore dei piani di accesso*.

L'ottimizzazione di una query (compito dell'ottimizzatore) richiede l'*enumerazione* dei possibili piani di accesso (ricordiamo che un piano di accesso è un modo concreto per risolvere le query, cioè è la sequenza di azioni effettivamente da eseguire) per la sua risoluzione e la *stima del costo* di ognuno di questi al fine di selezionare il piano dal costo minore stimato. La stima del costo di ogni piano di accesso viene effettuata dal valutatore dei piani di accesso usando stastiche fornite dai cataloghi. Si noti che un problema ancora in fase di studio è il seguente: il fatto che l'ottimizzatore si basa sulle stime del valutatore di piani, quale probabilità si ha che la scelta del piano di accesso risulti non essere ottimale ma sub-ottimale se la stima effettuata dal valutatore dei piani è errata? La risoluzione di questo problema non riguarda il corso che stiamo trattando, ma si voglia notare la grande connessione dei due moduli.

1.1 Ottimizzatore

Iniziamo subito presentando il funzionamento dell'ottimizzatore in DB2. La figura (Fig. 1) mostra le varie fasi che vengono attraversate. Ricordiamo dapprima che per poter eseguire al meglio una interrogazione ogni *dbms* utilizza un ottimizzatore che ha come compito speficifico di ricevere in input una data Query scritta in SQL (che ricordiamo essere un linguaggio procedurale, dunque non specifica effettivamente il modo di risolvere ciò che è richiesto) e da questo derivare un piano di accesso composto da operatori fisici e metodi di accesso ai dati: per fare questo lavoro l'ottimizzatore deve poter confrontare diversi piani di accesso, compito assegnato al valutatore dei pianidi accesso che utilizzando statistiche presenti nei *cataloghi* stiam il costo di ogni singolo operatore e dunque (sommando i costi) di ogni piano di accesso.

Figura 1:



Analizziamo in breve le varie fasi per poi discuterle in maniera più approfondita.

Avendo una Query da risolvere, la prima fase a cui viene sottoposta è la *Parse Query* che permette un controllo semantico e lessicale della stessa per vedere se la query è scritta "bene" oppure no. Subito dopo passa per il *Check*

Semantics, poichè anche se scritta sintatticamente bene, potrebbe essere errata a livello semantico come ad esempio se richiede di analizzare un attributo non esistente di una determinata relazione. La fase successiva è il *Rewrite Query* durante la quale la Query viene riscritta in un modo alternativo tentando di eliminare alcuni aspetti quali magari la ricorsione o le subquery. Il *Pushdown Analysis* serve per decidere, ad esempio, quali condizioni e predicati effettuare per prima, ad esempio se è presente una clausola WHERE locale applicabile prima di effettuare il join con una relazione. L'*Optimize Access Plan* è la prima vera fase di ottimizzazione (che insieme al *Remote SQL Generator* che non tratteremo, fase che serve a gestire dati distribuiti) che genera il piano di accesso ritenuto ottimale. Lo schema gira intorno al *Query Graph Model*, cioè un modello a grafo (più spesso sarà un albero) rappresentativo della query stessa (è una rappresentazione interna) in cui i nodi rappresentano i vari *operatori* mentre gli archi sono i *dati* a cui vengono applicati. Una volta trovato il piano di accesso ottimale viene generato del codice eseguibile attraverso la fase *Generate Excutable Code* il quale viene dunque eseguito durante la fase di *Execute Plan.* È possibile, inoltre, su DB2, visualizzare attraverso l'*Explain Table* (nella fattispecie attraverso il *Visual Explain*) il piano di accesso che è stato generato: viene data effettivamente la possibilità di controllarlo visivamente (o attraverso riga di comando con il *db2exfmt Tool*). Bisogna far particolare attenzione ai casi in cui l'ottimizzazione si basa su funzioni parametrici, cioè in cui l'ottimizzazione viene effettuata attraverso delle funzioni parametriche che vengono poi riempite con i dati effettivi passate dalla query: in questo caso, il *dbms* potrebbe generare diversi piani di accesso (e quindi non solo uno, cioè quello ottimale) proprio perchè a seconda del dato effettivamente passato, un piano di accesso potrebbe risultare migliore di un altro.

1.1.1 Parsing e check semantico

Il primo passo consiste nel verificare la validità lessicale e sintattica della query (*parsing*): al termine di questo passo viene prodotta una prima rappresentazione interna che evidenzia gli oggetti interessati (relazioni, attributi, ecc. . .). La fase di *check semantico* facendo uso dei *cataloghi* verificha che gli oggetti referenziati esistano effettivamente, controllando inolter che gli operatori siano applicati a dati di tipo opportuno e infine che l'utente abbia i privilegi necessari per eseguire le operazioni presente nella query stessa.

1.1.2 Riscrittura di query

Ha lo scopo di semplificare la query da ottimizzare. Le attivita di riscrittura della query sono diverse: risoluzione di viste, trasformazione di subquery in join, eliminazione di join ridondanti (ad esempio derivate da altre riscritture), spostamento/eliminazione di DISTINCT, pushdown dei predicati e trasformazione dei predicati OR in predicati IN.

Risoluzione di viste Per un semplice esempio si consideri la seguente vista

```
CREATE VIEW EmpSalaries(EmpNo, Last, First, Salary)
AS SELECT EmpNo, LastName, FirstName, Salary
   FROM Employee
   WHERE Salary>2000
```

e la seguente query che referenzia la vista prima indicata

```
SELECT Last, First
FROM EmpSalaries
WHERE Last LIKE 'B%'
```

La risoluzione della vista porta a riscrivere la query come

```
SELECT LastName, FirstName
FROM Employee
WHERE Salary>2000
AND LastName LIKE 'B%'
```

La riscrittura della query (che inizialmente refernziava la VIEW EmpSalaries), ora referenzia la relazione Employee su cui la VIEW si basava e nella riscrittura vengono implementati in AND il predicato presente sulla VIEW e il predicato di selezione della query stessa.

Query innestate Le subquery sono query SELECT usate nella clausola WHERE di un'altra query. Una subquery si dice *correlata* se fa riferimento ad attributi ("variabili") delle relazioni nel blocco "esterno" altrimenti è detta *costante*. In genere si producono piani di accesso per la subquery come se fossero query a parte (e quindi gestite in maniera autonoma): per evitare però di "perdere" dei piani di accesso a basso costo (e quindi estremamente utili), l'ottimizzatore cerca di riscrivere la query senza usare subquery facendola diventare "flat".

Vediamo un esempio di *Unnesting*. Il passaggio a una forma senza subquery alle volte è immediato: ad esempio la seguqne query:

```
SELECT EmpNo, PhoneNo
FROM Employee
WHERE WorkDept IN (
        SELECT DeptNo
        FROM Department
        WHERE DeptName='Operations'
)
```

viene riscritta come

```
SELECT EmpNo, PhoneNo
FROM Employee as E, Department as D
WHERE E.WorkDept = D.DeptNo
        AND D.DeptName = 'Operations'
```

In questo esempio la riscrittura trasforma la subquery in un join fra le due relazioni: in generale la clausola IN è di facile trasformazione, trasformazione non sempre banale invece in caso di IN negata (cioè NOT IN).

Un secondo esempio, in cui è presente il NOT IN, l'unnesting risulta essere più complicato. La query

```
SELECT E.EmpNo
FROM Employee AS E
WHERE E.EmpNo NOT IN (
        SELECT EA.EmpNo
        FROM Emp_Act as EA
)
```

viene riscritta usando un outer join

```
SELECT Q.EmpNo
FROM (
        SELECT E.EmpNo, EA.EmpNo
        FROM Emp_Act as EA RIGHT OUTER JOIN Employee E
        ON (E.EmpNo = EA.EmpNo)
) AS Q(EmpNo,EmpNo1)
WHERE Q.EmpNo1 IS NULL
```

Si pensi inizialmente che una prima riscrittura potrebbe portare ad avere una EXCEPT tra due relazioni, quella di "esterna" e quella "interna" (subquery) che porta ad uno stesso risultato. Come invece viene mostrato nell'esempio, in questo caso la riscrittura si avvale di RIGHT OUTER JOIN sulla relazione presente nella subquery, infatti con il RIGHT OUTER JOIN si andranno a inserire nel risultato delle tuple "dangling" (tuple che non fanno match sul predicato di join) che hanno dei valori NULL sugli attributi della relazione "esterna" Employee: attraverso il predicato IS NULL vengono selezionate esattamente queste ultime portando ad avere il risultato della query.

Il *dbms* può sfruttare dei vincoli (relazionali) presenti sul modello per semplificare e riscrivere le query: ad esempio se l'attributo *EmpNo* è attributo chiave della relazione Employee, allora la query

```
SELECT DISTINCT EmpNo
FROM Employee
```

può essere riscritta più semplicemente come

```
SELECT EmpNo
FROM Employee
```

poichè la clausola DISTINCT risulta essere superflua (facendo risparmiare dunque la richeista di eliminazione di duplicati poichè l'attributo essendo chiave né è privo).

Un ulteriore esempio di semplificazione basata sui vincoli è quella espressa dal vincolo di *Foreign Key*: se ad esempio l'attributo EmpNo della relazione Emp_Act ha il vincolo di *foreign key* con l'attributo EmpNo di Employee, allora la seguente query

```
SELECT EA.EmpNo
```

```
FROM Emp_Act as EA
WHERE EA.EmpNo IN (
        SELECT EmpNo
        FROM Employee
)
```

viene riscritta come

```
SELECT EA.EmpNo
FROM Emp_Act as EA
```

Se inoltre EA.EmpNo ammettesse valori nulli (che si vogliono eliminare per mantenere coerenza con la query richiesta), la riscrittura aggiungerebbe anche la clausola

```
WHERE EA.EmpNo IS NOT NULL
```

Vediamo infine un ultimo esempio di riscrittura che utilizza i vincoli sui privilegi delle viste. Si supponga di avere la seguente vista

```
CREATE VIEW People AS
        SELECT FirstName, LastName, DeptNo, MgrNo
        FROM Employee as E, Department as D
        WHERE E.WorkDept = D.DeptNo
```

e si esegua la seguente query

```
SELECT LastName, FirstName
FROM People
```

in cui, però, chi esegue la query non ha il privilegio di SELECT né su Employee né su Department. Allora il *dbms* può semplificare la query come segue

```
SELECT LastName, FirstName
FROM Employee
```

in cui ha sostituito la vista *People* con la relazione *Employee* (eliminando anche il join con Department poichè, per ipotesi, WorkDept ha un vincolo di *foreign key* su DeptNo) eventualmente aggiungendo la clausola

```
WHERE WorkDept IS NOT NULL
```

1.1.3 Rappresentazione interno della query

Una query, una volta riscritta, viene elaborata per avere una rappresentazione interna (utile al *dbms*): come molto spesso accade la rappresentazione interna più comoda da gestire è una *rappresentazione ad albero* (anche se, a volte, in caso di ricorsione si ricorre a grafi). Le *foglie* sono le *relazioni* presenti nella query, i *nodi interni* rappresentano gli *operatori dell'algebra* (estesa), i vari *rami* altro non sono che i *flussi dei dati* dal basso verso l'alto e infine la *radice* rappresenta il *risultato dell'interrogazione.*

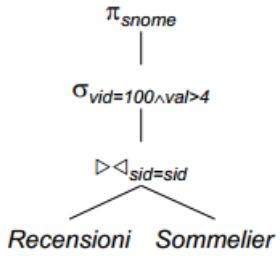
Si veda come la rappresentazione interna ad albero (Fig. 2) della seguente query

```
SELECT S.snome
FROM Recensioni as R, Sommelier as S
WHERE R.sid=S.sid AND R.vid=100 AND S.val>4
```

che in algebra relazionale viene espressa come

$$\pi_{snome}(\sigma_{vid=100 \wedge val>4}(Recesioni \triangleright\triangleleft_{sid=sid} Sommelier))$$

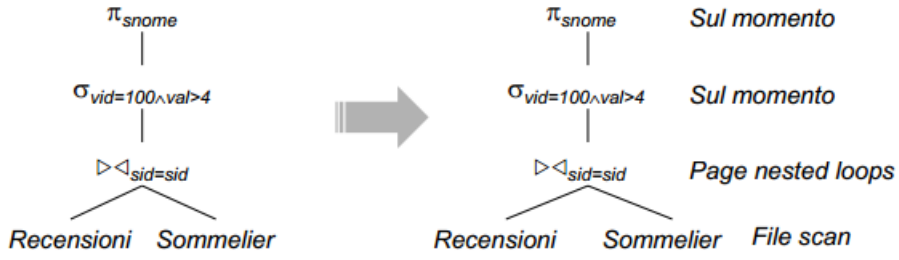
Figura 2:



1.2 Piano di accesso

Dalla prima rappresentazione ad albero (logico) mostrata al punto precedente, l'ottimizzatore trasforma (arricchisce) la rappresentazione ad albero della query in un *albero di operatori fisici*, in cui ora ogni operatore logico è sostituito da un operatore fisico che lo implementa, e le relazioni vengono contraddistinte anche con un metodo di accesso per la loro lettura. La figura (Fig. 3) mostra il passaggio.

Figura 3:



1.2.1 Esecuzione di piani di accesso

Per eseguire un piano di accesso occorre valutare gli operatori dell'albero a partire dal basso. Esistono due possibilità di esecuzione:

- Per *materializzazione* (non viene utilizzata poichè inefficente) in cui ogni operatore memorizza il proprio risultato in una tabella temporanea. Un operatore a livello più alto deve attendere che tutti gli operatori di input (cioè gli operatori a livello più basso) abbiano terminato la loro esecuzione per iniziare a produrre il proprio risultato. Si noti che ogni operatore produce un risultato "intermedio" che deve essere memorizzato su una tabella temporanea di dimensioni, spesso, decisamente maggiori delle dimensioni del risultato finale. Il modello è sicuramente semplice da implementare, ma il problema di dover gestire i risultati intermedi che potrebbero portare ad operazioni di I/O, fà sì che questo modello di esecuzione non venga quasi mai implementato.

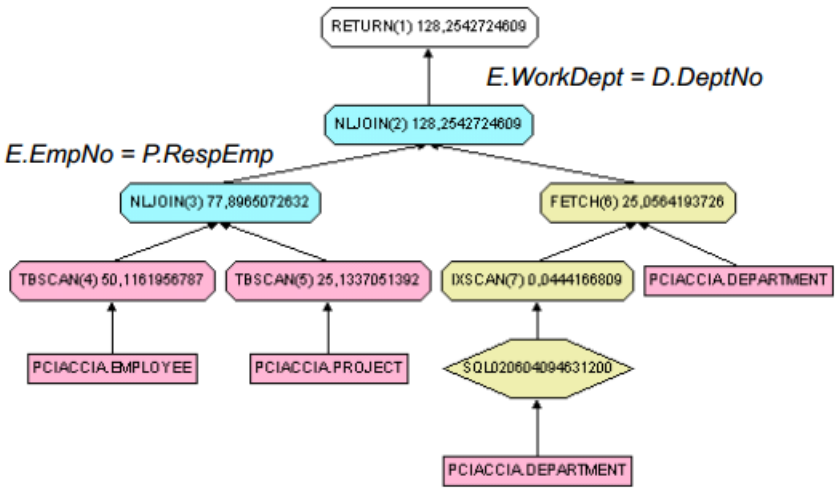
- In *pipeline* (tramite iteratori) in cui ogni operatore richiedere un risultato agli operatori di input e dunque può essere visto come una sorta di iteratore: di conseguenza un'operatore non è "bloccante" (cioè non deve attendere che il nodo a livello più basso abbia completato la sua esecuzione) ma d'altra parte non sempre è applicabile/possibile (ad esempio per *sort* in quanto per riuscire ad avere la prima tupla da poter passare al nodo genitore devo per forza aver scandito almeno tutta l'entità su cui sta elaborando).

Esecuzione per materializzazione Si prenda d'esempio la seguente query

```
SELECT P.ProjNo, E.EmpNo, D.*
FROM Department as D, Employee as E, Project as P
WHERE E.WorkDept = D.DeptNo AND E.EmpNo = P.RespEmp
```

che ha rappresentazione ad albero mostrata in figura (Fig. 4)

Figura 4:



in cui TBSCAN rappresenta una *scansione sequenziale*, NLJOIN rappresenta un join effettuato tramite *nested loops* e IXSCAN rappresenta una *index nested loops*. L'esecuzione per materializzazione produce come primo risultato il risultato del primo join (Fig. 5)

Figura 5:

PROJNO	EMPNO	WORKDEPT
AD3100	000010	A00
MA2100	000010	A00
PL2100	000020	B01
IF1000	000030	C01
IF2000	000030	C01
OP1000	000050	E01
OP2000	000050	E01
MA2110	000060	D11
AD3110	000070	D21
OP1010	000090	E11
OP2010	000100	E21
MA2112	000150	D11
MA2113	000160	D11
MA2111	000220	D11
AD3111	000230	D21
AD3112	000250	D21
AD3113	000270	D21
OP2011	000320	E21
OP2012	000330	E21
OP2013	000340	E21

e solo a partire da tale risultato "intermedio" si può calcolare il secondo join che produce il seguente risultato (Fig. 6)

Figura 6:

PROJNO	EMPNO	DEPTNO	DEPTNAME	MGRNO	ADMRDEPT	LOCATION
AD3100	000010	A00	SPIFFY CO...	000010	A00	
MA2100	000010	A00	SPIFFY CO...	000010	A00	
PL2100	000020	B01	PLANNING	000020	A00	
IF1000	000030	C01	INFORMAT...	000030	A00	
IF2000	000030	C01	INFORMAT...	000030	A00	
OP1000	000050	E01	SUPPORT ...	000050	A00	
OP2000	000050	E01	SUPPORT ...	000050	A00	
MA2110	000060	D11	MANUFAC...	000060	D01	
AD3110	000070	D21	ADMINIST...	000070	D01	
OP1010	000090	E11	OPERATIO...	000090	E01	
OP2010	000100	E21	SOFTWAR...	000100	E01	
MA2112	000150	D11	MANUFAC...	000060	D01	
MA2113	000160	D11	MANUFAC...	000060	D01	
MA2111	000220	D11	MANUFAC...	000060	D01	
AD3111	000230	D21	ADMINIST...	000070	D01	
AD3112	000250	D21	ADMINIST...	000070	D01	
AD3113	000270	D21	ADMINIST...	000070	D01	
OP2011	000320	E21	SOFTWAR...	000100	E01	
OP2012	000330	E21	SOFTWAR...	000100	E01	
OP2013	000340	E21	SOFTWAR...	000100	E01	

Esecuzione in pipeline Mostriamo subito un esempio. Si inizia ad eseguire il primo join il quale produrrà la prima tupla ("intermedia") mostrata in figura (Fig. 7)

Figura 7:

PROJNO	EMPNO	WORKDEPT
AD3100	000010	A00

la quale viene subito passata in input al secondo join che inizia la ricerca di *matching tuples* producendo la prima tupla del risultato mostrata in figura (Fig. 8)

Figura 8:

PROJNO	EMPNO	DEPTNO	DEPTNAME	MGRNO	ADMRDEPT	LOCATION
AD3100	000010	A00	SPIFFY CO...	000010	A00	

L'esecuzione prosegue cercando eventuali ulteriori match per la tupla prodotta dal primo join (si noti che nel frattempo il primo join è in attesa poichè è richiesta l'elaborazione ora del secondo): quando è terminata la scandione della relazione interna (Department), il secondo join richiede al primo di produrre un'ulteriore tupla e il procedimento si itera.

Poichè l'esecuzione in *pipeline* di un operatore è indipendente dal tipo di operatore è necessario sapere quando effettivamente "passare" una tupla da un nodo ad un altro (il sistema è demand drive): ecco che si presenta l'esigenza di un'interfaccia. Per semplificare il codice di coordinamento dell'esecuzione dei *piani di accesso* viene implementata un'interfaccia standard per ogni tipo di operatore cioè ogni operatore deve implementare dei *metodi* comuni per l'esecuzione dell'algoritmo

open inizializza e alloca il buffer, passa alcuni parametri (ad esempio gli attributi necessari nel SELECT) e richiama *open* sui figli ricorsivamente

hasNext verifica se ci sono ulteriori tuple

next restituisce la prossima tupla

reset riparte dalla prima tupla (necessario ad esempio nei *nested loops*)

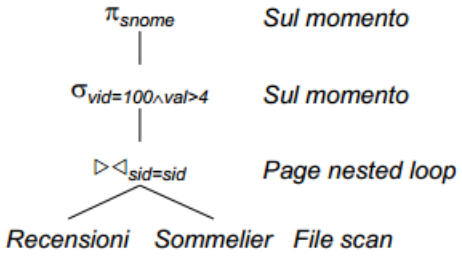
close termina e rilascia le risorse

Si noti che l'algoritmo di esecuzione dei piani di accesso parte dalla radice, la quale partendo con il metodo *open* andrà, in maniera ricorsiva, a far chiamare il metodo *open* ai figli il quale lo faranno chiamare ai figli, etc.. La radice, a questo punto, richiede la prima tupla che apparterà al risultato tramite il metodo *next*, il quale andrà a richiedere la generazione di una tupla ai figli, i quali a loro volta (a meno che non siano già delle relazioni) richiederanno ai loro figli una tupla da poter elaborare e così via. L'algoritmo terminata quando il metodo *hasNext* chiamato dalla radice risulterà avere valore *false.*

1.2.2 Piani di accesso alternativi

Vediamo come, una stessa query, possa avere un numero svariato di piani di accesso alternativi. Supponiamo di dover eseguire l'albero mostrato in precedenza che qui rimostriamo (Fig. 9)

Figura 9:

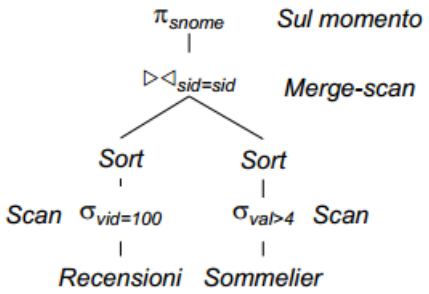


Il costo dell'operazione è dato dal costo del join: 501'000 operazioni I/O ovvero circa 1.4 ore. Ma si potrebbe fare decisamente di peggio, ad esempio con

un *prodotto cartesiano* tra le relazioni con successiva *selezione* sull'attributo di join. Vogliamo dunque cercare di ridurre il costo evitando di valutare però tutte le possibili alternative.

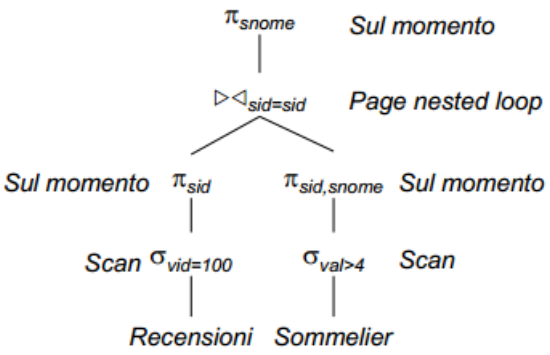
Poichè il join è un'operazione costosa una prima soluzione è quella di diminuire la dimensione degli input: una possibilità è quella di applicare le selezioni *prima* di effettura il join (effettuare dunque un *push-down* dell'operazione di selezione). Ad esempio la selezione *vid=100* si applica solo a Recensioni mentre la selezione *val>4* si applica solo a Sommelier: vengono dunque create delle relazioni temporanee T1 e T2 che soddisfano le selezioni che successivamente vengono ordinate (per poter eseguire il *mergescan*). Il costo complessivo è dato da $Costo = P(R)+P(T1)+P(S)+P(T2)+sort(T1)+sort(T2)+P(T1)+P(T2) = 1000+10+500+250+40+1500+10+250 = 3560$ (35 secondi). Si veda come l'albero rappresentante la query si sia modificato (Fig. 10)

Figura 10:



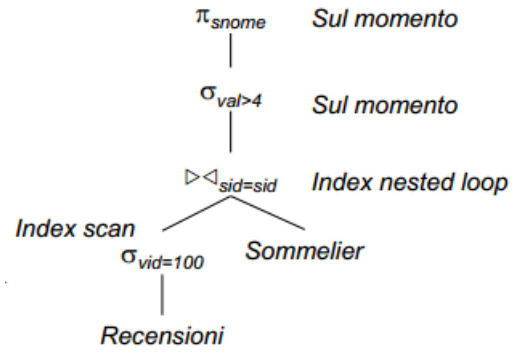
Un'altra possibilità invece è quella di introdurre una proiezione: infatti sia per il join che per l'output sono richiesti solo gli attributi *sid* e *snome*. Questo porta ad avere tabelle temporanee di dimensioni ridotte. Possiamo usare ad esempio il *page nested loops* e dunque il costo risultante è dato da $Costo = P(R)+P(T1')+P(S)+P(T2')+P(T1')+P(T2') = 1000+3+500+250+3+250 = 2006$ (20 secondi). Ancora una volta l'albero viene modificato come mostrato in figura (Fig. 11)

Figura 11:



Infine la presenza di *indici* può ridurre ulteriormente il costo. Supponiamo, ad esempio, di avere un indice hash su R.vid e uno su S.sid. Possiamo utilizzare dunque l'*index nested loops* il che porta il costo ad essere $costo = L(R) + f \cdot N(R) \cdot 1 + 10 + 1000 = 1010$ (10 seco). Si noti che la selezione su S.val di Sommelier non può essere *pushed* poichè per effettuare un index nested loops l'attributo dell'indice deve essere lo stesso dell'attributo di join, ma in questo caso l'indice è presente su S.val e non su S.vid (che sarebbe l'attributo di join necessario per l'index nested loops). L'albero modificato risulta essere quello mostrato in figura (Fig. 12)

Figura 12:



È evidente che nella riscrittura di un piano di accesso occorre che l'ottimizzatore non modifichi la semantica della query originale (cioè che i risultati a cui si perviene rimangano sempre gli stessi): occorre dunque conoscere l'*equivalenza delle espressioni* in algebra relazionale, cioè come un operatore può essere riscritto attraverso un altro operatore mantenendo però la stessa semantica. Si noti che bisogna anche ricordare che nell'equivalenza si devono implementare anche i valori NULL e i valori duplicati. Le regole più comuni per l'equivalenza

di espressioni nella modifica dei piani di accesso sono il *raggruppamento e commutatività* della *selezione* e della *proiezione*, la *commutatività e associatività* dell'operatore di join, il *push-down* dell'operatore di selezione sull'operatore di join...

1.3 Stima del costo del piano di accesso

Per ogni nodo dell'albero fisico occorre saper stimare il *costo*, la *dimensione del risultato* e l'*ordinamento* (se presente o meno) *del risultato e dei risultati parziali* (è importante saperlo poichè alcuni algoritmi cambiano il loro costi a seconda che i risultati siano ordinati o meno). Per il *costo* si utilizzano le stime di cui abbiamo già discusso nel precedente capitolo, tenendo conto che il *pipelining* permette di risparmiare il costo di scrittura e lettura dei risultati intermedi, cioè delle tabelle temporanee di output dei nodi interni. Precisiamo inoltre che riuscendo a stimare il costo di ogni singolo operatore, la composizione di essi riesce a stimare il costo del piano di accesso associato. Il tutto si basa su statistiche che vengono mantenuta aggiornate dal *dbms* in maniera automatica o sotto richiesta del *DB Administrator*.

In DB2 le statistiche, come già sappiamo, vengono mantenute all'interno dei cataloghi presenti nello *schema SYSSTAT*. Esse vengono aggiornate tramte il comando **RUNSTATS** e comprendo statistiche su tabelle, indici, colonne, gruppi di colonne, distribuzioni...

- **Esempi di Statistiche su tabelle**

  FPAGES Numero di pagine usate da una tabella

  NPAGES Numero di pagine contenenti record ($\leq FPAGES$ poichè a causa di eliminazioni potrebbero esserci pagine vuote non ancora compattate che però non contengono record della tabella)

  CARD Cardinalità di una tabella, ovvero il numero dei suoi record

- **Esempi di Statistiche su colonne**

  COLCARD Cardinalità dell'attributo (ovvero la stima dei valori distinti dell'attributo)

  AVGCOLLEN Dimensione media dell'attributo

  HIGH2KEY, LOW2KEY Secondo valore maggiore e minore dell'attributo

  NUMNULLS Numero dei valori NULL

- **Esempi di Statistiche su indici**

  NLEAF Numero delle foglie dell'indice

  NLEVELS Numero dei livelli dell'indice, cioè la sua altezza

CLUSTERRATIO Grado di clusterizzazione dei dati rispetto all'indice. Si noti che non esistono solo indici *clustered* e *unclustered*. Si prenda ad esempio una relazione di Studenti ordinata per Matricola: un indice su Matricola risulterà sicuramente *clustered*, ma un un indice su Età (dello studente) risulterà "abbastanza" clustered su Studenti poichè c'è una forte correlazione tra il numero di matricola e l'età dello studente: *clusterratio* dà dunque una stima del livello di clusterizzazione dell'indice

DENSITY Percentuale di foglie memorizzate sequenzialmente

NUMRIDS Numero di RID memorizzati nell'albero

1.3.1 Stime del numeri di valori distinti

È spesso necessario conoscere il numero di valori distinti di un attributo (o combinazione di attributi). L'approccio basato su *sorting* richiede tempo $O(P \cdot \log P)$ per ogni attributo. Esiste però un metodo molto più veloce basato su *hashing* noto come *linear counting* che ha complessità $O(P)$.

Per effettuare il **linear counting** si predispone in RAM un array B di grandezza BX. L'array B è un array di soli bit e viene inizializzato a 0. Vogliamo ora sapere il numero di valori distingi per un attributo A della relazione R. Si scandisce l'intera relazione R e per ogni tupla r si applica una funzione hash H a valori in $[0, B-1]$ all'attributo A di interesse. Si pone dunque $B[H(r.A)] = 1$ cioè si pone a 1 il bit dell'array risultante dalla funzione hash associata: si noti che in caso di valori duplicati non vi è alcun problema poichè essi andranno tutti a finire nella stessa posizione dell'array. Alla fine della scansione della relazione, si contano il numero Z di bit rimasti a 0. La stima del numero di valori distinti di A, indicata con NK(R.A), è data da

$$NK(R.A) \approx BX \cdot \ln\left(\frac{BX}{Z}\right)$$

Precisiamo che al termine dell'esecuzione i bit dell'array che hanno valore 1 sono bit che sono stati colpiti dalla funzione hash almeno una volta, mentre quelli che hanno valore 0 sono posizioni che la funzione hash non ha mai generato. Si noti inoltre che il numero di zeri Z non può essere uguale a 0 (altrimenti la formula risulta impossibile): bisogna garantire che $Z > 0$ (e per far ciò si può porre $BX = N$, tanto essendo un array di bit non occupperà molto spazio). Inoltre, da come si vede dalla formula, il numero delle tuple N della relazione (valore di input) non è una variabile del problema, risulta essere ininfluente, questo proprio per il motivo prima spiegato sulla presenza di duplicati o meno.

Come si è arrivati alla formula sulla stima? Viene seguito un procedimento analogo al modello di Cardennas:

1. $\frac{1}{BX}$ equivale alla probabilità che una determinata posizione dell'array B venga generata da un "lancio" della funzione hash
2. $1 - \frac{1}{BX}$ equivale alla probabilità che una determinata posizione non venga generata da un lancio

3. $\left(1 - \frac{1}{BX}\right)^{NK}$ equivale alla probabilità che una determinata posizione non venga generata da NK lanci

4. $1 - \left(1 - \frac{1}{BX}\right)^{NK}$ equivale alla probabilità che una determinata posizione venga generata almeno una volta in NK lanci

5. $BX \cdot \left[1 - \left(1 - \frac{1}{BX}\right)^{NK}\right]$ equivale alla probabilità che una qualsiasi posizione (ho moltiplicato per BX) venga generata almeno una volta in NK lanci

Di conseguenza data l'array B di dimensione BX, il numero di "uni" presenti alla fine del procedimento equivale a $BX - Z$ (numero totale meno il numero di zeri) e questo equivale proprio a

$$BX - Z = BX \cdot \left[1 - \left(1 - \frac{1}{BX}\right)^{NK}\right]$$

che altro non è il punto 5 del procedimento prima analizzato. Attraverso alcuni passaggi aritmetici otteniamo

$$\left(1 - \frac{1}{BX}\right)^{NK} = \left(1 - \frac{1}{BX}\right)^{BX \cdot \left(\frac{NK}{BX}\right)} \sim e^{-\frac{NK}{BX}}$$

andando a sostituire otteniamo dunque

$$BX - Z \approx BX \cdot \left(1 - e^{-\frac{NK}{BX}}\right) = BX - BX \cdot e^{-\frac{NK}{BX}}$$

e quindi

$$Z = BX \cdot e^{-\frac{NK}{BX}} \rightarrow \ln\left(\frac{Z}{BX}\right) \approx -\frac{NK}{BX}$$

e dunque

$$NK \approx BX \cdot \ln\left(\frac{BX}{Z}\right)$$

Il metodo è ovviamente applicabile a più attributi, o combinazione di essi, anche applicato in parallelo. Dunque è un calcolo veloce per avere una stima dei valori distinti.

1.3.2 Istogrammi

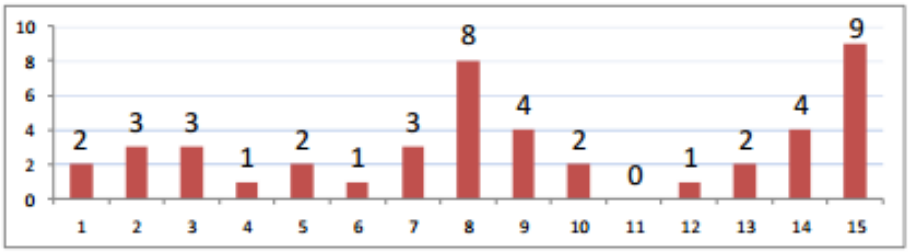
Il solo numero di valori distinti può portare a stime poco accurate che possono quindi influenzare in modo impredicibile il processo di ottimizzazione. Ad esempio dati N dipendenti e NK(ruolo) ruoli: quanti dipendenti hanno il ruolo di 'operaio'? Secondo la stima prima descritta $\frac{N}{NK(ruolo)}$. Quanti dipendenti hanno invece il ruolo di 'direttore di filiale'? Sempre $\frac{N}{NK(ruolo)}$. Come si vede la stima descritta presuppone una distribuzione uniforme, la quale, spesso, è

molto lontana dalla realtà dei dati. Si ha bisogno dunque di informazioni più dettagliate sulla distribuzione dei valori: ecco che praticamente tutti i dbms si basano e implementano *istogrammi.*

Esistono diversi tipi di istogrammi: *Equi-width, Equi-depth* e *Compressed.*

Partendo da un'esempio in cui la distribuzione delle N=45 tuple con NK=15 valori distinti è data dalla seguente figura (Fig. 13)

Figura 13:



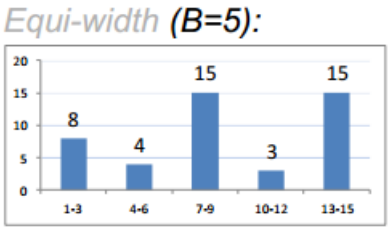
descriviamo i varii istogrammi vedendo come poi implementano la loro distribuzione all'esempio qui sopra descritto.

Equi-width Nell'istogramma *equi-width* il dominio viene suddiviso in B intervalli della stessa ampiezza, cioè preso ad esempio un range di valori da 0 a 100, lo possiamo suddividere in B=10 intervalli di ampiezza 10 ottenendo [0,10], [10,20], [20,30], ...

Il problema di questo modello è che risulta molto poco accurati, non suddivide l'intervallo dei dati in maniera "dipendente" dalla vera distribuzione: ecco che non vi è alcuna garanzia sull'errore che si può commettere.

La figura (Fig. 14) mostra l'istogramma *equi-width* applicato all'esempio prima citato.

Figura 14:



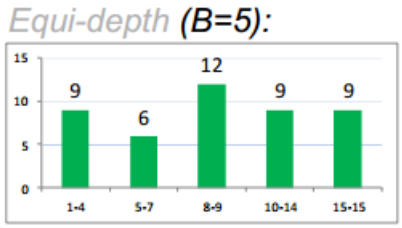
Equi-depth Nell'istogramma *equi-depth* il dominio viene suddiviso in B intervalli in modo che il numero di tuple in ogni livello sia circa lo stesso, cioè se

preso preso il nostro dominio lo dividiamo in 4 parti, vogliamo, intuitivamente, che ogni intervallo contenga circa il 25% delle tuple.

Il problema di questo modello, che risulta decisamente migliore dell'*equi-width*, si ha in presenza di molti valori frequenti causando dei "picchi" su alcuni intervalli.

La figura (Fig. 15) mostra l'istogramma *equi-depth* applicato all'esempio prima citato.

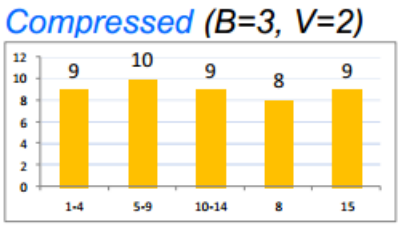
Figura 15:



Compressed L'istogramma *Compressed* è equivalente all'istogramma *equi-depth*, solo che per ognnuno dei V valori più frequenti, viene mantenuto un contatore separato. L'istogramma compressed risulta essere il più usato fra i *dbms*.

La figura (Fig. 16) mostra l'istogramma *compressed* applicato all'esempio prima citato.

Figura 16:



1.4 Query

Tutto ciò che abbiamo introdotto finora è finalizzato all'ottimizzazione delle query. Vediamo dunque come è possibile gestire una **Query su singola relazione**.

1.4.1 Su Singola relazione

Se la query contiene un'unica relazione nella clausola FROM dobbiamo valutare solo le *proiezioni* e le *selezioni* più eventuali *raggruppamenti* e *operazioni aggregate*.

Esistono quattro possibili soluzioni:

1. Scansione sequenziale
2. Uso di un solo indice (eventualmente clustered)
3. Uso di più indici
4. Uso solo di un indice (non si accede ai dati, il caso INDEX ONLY)

Vediamo i diversi casi applicati ad un esempio pratico. Vogliamo risolvere la seguente query su singola relazione

```
SELECT rivista
FROM Recensioni
WHERE vid=417 AND anno>2005
```

Sappiamo che esiste un indice hash sull'attributo *vid* di Recensioni, un indice B+tree sull'attributo *anno* e un indice B+tree su (*vid, anno, rivista*)

Scansione sequenziale Con la scansione sequenziale si leggono tutte le pagine della relazione Recensioni e si effettuano selezione e proiezione sul momento, dunque il costo è dato da $costo = P(R)$ (numero di pagine della relazione).

Uso di un solo indice Esistendo diversi indici, scegliamo l'indice più selettivo (cioè quello che "taglia" più tuple) che risulta essere l'indice su *vid*. Si leggono dunque i record tramite le RID che l'indice su vid fornice e infine sulle tuple lette viene valutato il predicato su *anno* ed effettuata la proiezione sull'attributo *rivista*.

Uso di più indici Si usa sia l'indice su *vid* sia l'indice su *anno*. Le RID ottenute da entrambi gli indici si intersecano e si leggono dunque i record associate alle RID presenti nell'intersezione: infine si effettua la proiezione su *rivista*.

Uso solo di un indice (INDEX ONLY) Essendo presente un indice B+tree su (vid, anno, rivista) è possibile ricavare tutti i dati del risultato senza accedere effettivamente ai dati. Si utilizza dunque l'indice composto per selezionare i "record" utili e restituendo solo il valore dell'attributo rivista.

L'**ottimizzatore** in DB2 "classifica" i predicati presenti nella query in quattri classi (la lista è in ordine di efficienzza decrescente):

1. **Range-delimiting:** i predicati *range-delimiting* delimitano il range delle foglie dell'indice a cui accedere (ad esempio il predicato WHERE vid=4 finisce in questa categoria)

2. **Index SARGable:** i predicati *index SARGable* non delimitano il range di foglie, ma riducono solo attraverso l'indice il numero di dati che corrispondono alle RID che generiamo. Se ad esempio abbiamo un indice su (A,B,C) e il predicato è WHERE $A = 5\, AND\, C > 10$, il predicato *C>10* non delimiterà il numero di foglie a cui bisogna accedere all'indice perchè non è un prefisso della chiave con l'indice è generato, ma nonostante questo si può utilizzare l'indice su (A,B,C) per ottenere direttemente un numero inferiore di RID a cui bisognerà poi accedere: in pratica la condizione C>10 viene valutata sull'indice e non sui dati. In questo caso, dunque, C>10 è un predicato *Index SARGable*

3. **DATA SARGable:** i predicati *data SARGable* sono predicati utilizzabili all'istante dell'accesso ai dati

4. **Residual:** sono tutti i predicati rimanenti, ad esempio subquery correlate, il predicato ANY, il predicato ALL, ...

La tabella (Tab. 1) riassume gli effetti dei 4 tipi di predicati.

Tabella 1:

	Range-Delimiting	**Index SARGable**	**Data SARGable**	**Residual**
Riduzione Index I/O	Sì	NO	NO	NO
Riduzione Data I/O	Sì	Sì	NO	NO
Riduzione num. tuple	Sì	Sì	Sì	NO
Riduzione output finale	Sì	Sì	Sì	Sì

Ricordiamo infine che i metodi di accesso in DB2 ai dati sono per *scansione sequenziale*, per scansione tramite *indice B+tree*, per scansione tramite *lista di RID* ottenuta per unione o intersezione (ma non entrambe) da 2 o più indici. L'accesso tramite indice può avvenire per test di predicati come IS (NOT) NULL o predicati di uguaglianza o uguaglianza con una costante..., oppure tramite ordinamento.

1.4.2 Su più relazioni

In caso di più relazioni, il predicato FROM contiene più di una relazione. Il primo problema da affrontare dunque è come effettuare il join (a livello logico) delle N relazioni coinvolte.

Il numero possibile di modi per effettuare il join (includendo i prodotti cartesiani, poichè se ho un join fra A1 e A2 e un join fra A2 e A3 allora fra A1 e A3 esiste una relazione di prodotto cartesiano e non di join) è dato da

$$\frac{(2 \cdot (N-1))!}{(N-1)!} = N! \cdot \frac{\binom{2 \cdot (N-1)}{(N-1)}}{N}$$

in cui $N!$ altro non è il numero di permutazioni delle N relazioni, mentre il secondo fattore è il numero di modi in cui è possibile "mettere le parentesi" e dunque di attribuire delle priorità a dei join piuttosto che ad altri.

Si noti che il numero di join cresce fattorialmente:

- 2 modi avendo 2 relazioni
- 12 modi avendo 3 relazioni
- 120 modi avendo 4 relazioni
- 1680 modi avendo 5 relazioni
- 30'240 modi avendo 6 relazioni
- 665'280 modi avendo 7 relazioni
- 17'297'280 modi avendo 8 relazioni

Date ad esempio 3 relazioni R1, R2 ed R3, i 12 possibili modi di combinare i join sono

1. $(R1 \triangleright\triangleleft R2) \triangleright\triangleleft R3$
2. $(R2 \triangleright\triangleleft R1) \triangleright\triangleleft R3$
3. $(R1 \triangleright\triangleleft R3) \triangleright\triangleleft R2$
4. $(R3 \triangleright\triangleleft R1) \triangleright\triangleleft R2$
5. $(R2 \triangleright\triangleleft R3) \triangleright\triangleleft R1$
6. $(R3 \triangleright\triangleleft R2) \triangleright\triangleleft R1$
7. $R1 \triangleright\triangleleft (R2 \triangleright\triangleleft R3)$
8. $R2 \triangleright\triangleleft (R1 \triangleright\triangleleft R3)$
9. $R1 \triangleright\triangleleft (R3 \triangleright\triangleleft R2)$
10. $R3 \triangleright\triangleleft (R1 \triangleright\triangleleft R2)$
11. $R2 \triangleright\triangleleft (R3 \triangleright\triangleleft R1)$
12. $R3 \triangleright\triangleleft (R2 \triangleright\triangleleft R1)$

1.5 Determinazione piano ottimale

L'ottimizzatore dispone di una *strategia di enumerazione* dei piani di accesso i cui compiti principali sono *enumerare* tutti i piani che potenzialmente possono risultare ottimali senza generare i piani di accesso che sicuramente non possono risultare tali. Con questo si intende una strategia algoritmica di creazione di ogni singolo piano di accesso, cioè, se prendiamo l'esempio sui join, la strategia potrebbe ad esempio per prima cosa generare tutte le possibili N! permutazioni e successivamente per ogni permutazione mettere le parentesi.

Lo spazio dei possibili piani di accesso è, tipicamente, molto vasto (esponenziale in N, si veda anche solo un join fra 8 relazioni), infatti non solo ogni

operatore può essere realizzato in più modi, inoltre ogni query può essere riscritta in modi diversi ma equivalenti ed infine possono esistere diverse strutture che agevolino l'accesso ai dati e dunque per accedere ad ogni relazione possono esistere più modi. Per questo motivo, in genere, vengono applicate regole *euristiche* per ridurre lo spazio di ricerca: da questo si conclude che l'ottimo non è garantito (sono regole euristiche) ma generalmente si trova una soluzione valida in tempo ragionevole. Una tipica regola euristica, adottata praticamente sempre, è la non generazione dei piani di accesso che prevedono l'utilizzo di prodotti cartesiani.

Si noti che diversi *dbms* mettono a disposizione strumenti che permettono di controllare esplicitamente i tipi di piani considerati, ad esempio DB2 permette di impostare un parametro sul livello di ottimizzazione (caso tipico è il livello 5 in cui i prodotti cartesiani vengono scartati) che va dal livello 0 al livello 9.

1.5.1 Left-deep trees

Notiamo per prima cosa che data una sequenza di operazioni, nell'esempio che mostriamo una sequenza di join, è possibile generare diversi alberi a seconda di come vengono inserite le parentesi. Si vedano i due esempi in figura che mostrano gli alberi di un'ipotetica sequenza $R1 \triangleright\triangleleft R2 \triangleright\triangleleft R3$ con i due possibili modi di combinare le parentesi $(R1 \triangleright\triangleleft R2) \triangleright\triangleleft R3$ (Fig. 17) e $R1 \triangleright\triangleleft (R2 \triangleright\triangleleft R3)$ (Fig. 18).

Figura 17:

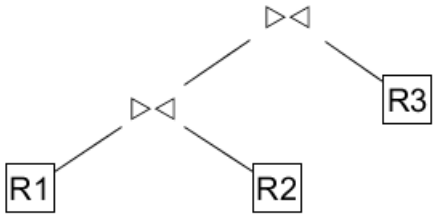
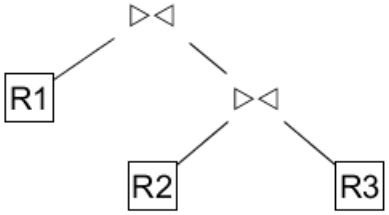


Figura 18:



Una regola euristica per ridurre lo spazio di ricerca, che spesso viene applicata, è data dall'utilizzo dei soli *left-deep trees* (un esempio è mostrato in Fig. 17),

ovvero alberi di join in cui il figlio destro di un nodo è sempre una relazione del database: gli alberi che non rispettano questa caratteristica vengono chiamati *bushy* (un esempio è mostrato in Fig. 18). Tentiamo di spiegarci meglio, quando diciamo che il figlio destro di un nodo deve essere una relazione del database si intende che effettivamente i dati sono reperibili da un predicato monotabellare: questo implica che è possibile che la relazione possa avere dei predicati locali come una selezione o una proiezione, ma sicuramente non sarà mai frutto di un join poichè l'output uscente da un join non è "una relazione del database". Inoltre, come detto precedentemente, ogni albero (ricordiamo che l'albero corrisponde ad una rappresentazione di un determinato piano di accesso) contenente un prodotto cartesiano non viene generato.

Il figlio destro di ogni nodo, come abbiamo detto, è una relazione: questa risulta essere la *relazione interna* che è obbligatoriamente materializzata cioè i cui risultati sono reperibili immediatamente poichè non derivano da un join. Si voglia precisare che sarebbe possibile anche avere un join nel figlio destro, ma ciò non rispecchia l'euristica del left-deep tree: infatti potrebbe anche essere possibile generare "al momento" il join del figlio destro; non per forza questo risultato può risultare sempre penalizzante, ma si noti che, solitamente, se il figlio sinistro è una relazione qualsiasi che richiede il join con il figlio destro, il figlio destro deve essere "ripetuto" più volte poichè i join richiedono di controllare più volte la relazione interna (che è rappresentata dal figlio destro), di conseguenza il dover riconsiderare (il che vuol dire ricalcolare) ogni volta il join del figlio destro per la valutazione del join con il figlio sinistro potrebbe essere molto costoso (ultima precisazione: si potrebbe pensare di utilizzare una tabella temporanea per salvare il join del figlio destro, ma questo implicherebbe l'utilizzo dell'esecuzione per materializzazione e non per pipeline). In questo modo è possibile sempre generare piani in *pipeline*. La soluzione dei *left-deep trees* è la soluzione utilizzata nel dbms *System R*.

L'utilizzo dei *left-deep trees* riduce, in parte, il numero dei piani di accesso generabili: infatti per ogni permutazione dei join esiste una solo modo di disporre le parentesi per costruire effettivamente un left-deep trees. Ad esempio avendo la sequenza $R1 \triangleright\triangleleft R2 \triangleright\triangleleft R3 \triangleright\triangleleft R4$ l'unico modo per generare un *left-deep trees* è disporre le parentesi nel seguente modo $((R1 \triangleright\triangleleft R2) \triangleright\triangleleft R3) \triangleright\triangleleft R4$

Vediamo un esempio: si prenda la seguente query

```
SELECT S.snome, V.vnome, R.anno
FROM Sommelier as S, Recensioni as R, Vini as V
WHERE S.sid=R.sid AND R.vid=V.vid
        AND R.rivista="Sapore DiVino"
```

Le possibili sequenze di *join* che risultano essere left-deep trees sono solo 4: infatti i possibili modi di combinare 3 relazioni come abbiamo visto sono 12, ma di queste sei risultano essere *alberi bushy* e infine altre due risultano avere dei *prodotti cartesiani* e dunque $4 = 12 - 6\,bushy - 2\,cartesiani$

1. $(R \triangleright\triangleleft S) \triangleright\triangleleft V$
2. $(S \triangleright\triangleleft R) \triangleright\triangleleft V$

3. $(R \triangleright\triangleleft V) \triangleright\triangleleft S$
4. $(V \triangleright\triangleleft R) \triangleright\triangleleft S$

1.5.2 Proprietà di piani di accesso

Il primo modo di operare per determinare il piano di accesso migliore è di enumerare i piani di accesso e per ognuno di essi valutarne il costo appliccando tutti i possibili algoritmi fisici per l'implementazione di un determinato operatore logico: ci si rende subito conto che questo modo ingenuo non può portare ad ottenere il risultato in maniera veloce. Un modo più furbo di operare si basa sul concetto di "dominazione" che introdurremo a breve: grazie ad esso la generazione dei piani di accesso da $O(N!)$ diverrà $O(2^N)$.

ootiamo che pgni nodo di un piano di accesso può anche essere visto come la radice di un piano di accesso "parziale" o "completo" e che ogni piano di accesso è caratterizzato da determinate *proprietà* quali*:*

1. Costo (il costo per eseguire il piano di accesso)
2. Cardinalità (il numero di tuple del risultato)
3. Schema (quali attributi genera il risultato del piano di accesso)
4. Relazioni (su quali relazioni lavora il piano)
5. Predicati applicati (che predicati sono stati applicati durante l'esecuzione del piano di accesso)
6. Ordine (l'ordine con cui vengono restituite le tuple in output)
7. ...

La *proprietà di un nodo* si può pensare come funzione delle proprietà dei figli e della proprietà dell'operatore del nodo stesso $\rightarrow$ $proprietà\, nodo = f(proprietà\, figli, operatore\, nodo)$ infatti, ad esempio, il costo di un nodo padre è sicuramente maggiore del costo dei nodi figli poichè sarà somma di questi ultimi più il costo dell'operatore del nodo padre.

Ordini significativi Tra tutte le proprietà di un piano di accesso, l'ordine risulta di estrema importanza, infatti come sappiamo potrebbe influire sui costi e implementazioni di determinati algoritmi.

L'ordine delle tuple di un nodo è detto *significativo* se può influenzare le operazioni ancora da compierere o il risultato finale. Prendiamo ad esempio la seguente query

```
SELECT S.snome, V.vnome, R.anno
FROM Sommelier as S, Recensioni as R, Vini as V
WHERE R.sid=S.sid AND R.vid=V.vid
        AND V.vnome="Merlot"
ORDER BY S.snome, V.cantina
```

dopo il join tra V e R gli ordini significativi sono:

1. *sid* infatti può influenzare il join con S (ad esempio con un merge-scan che richiede l'ordinamento dell'attributo di join)
2. *snome* poichè semplifica l'ORDER BY
3. *snome, cantina* poichè risolve completamente l'ORDER BY

ma non risulta in alcun modo significativo il solo ordine su *cantina* (poichè non aiuta nell'ORDER BY) o su *vid* (poichè non è di utilità nel join rimanente con S).

1.5.3 Pruning di piani parziali

La seguente osservazione è alla base della determinazione efficiente del piano ottimale:

Un piano di accesso (parziale) AP per i relazioni R_1, R_2, ..., R_i, il cui risultato è ordinato secondo un ordine O e il cui costo è maggiore di quello di un piano AP' per le stesse relazioni e con lo stesso ordine (o più significativo), non può essere esteso in un piano di accesso a costo minimo (cioè AP non deve essere considerato ed espanso). Si dice che AP' "domina" AP cioè che le proprietà di AP' dominano quelle di AP.

Tentiamo di capire meglio: prendiamo due piani di accesso AP e AP' e dimentichiamoci dell'ordine O. Supponiamo che il costo di AP sia superiore al costo di AP' (ricordiamo che il costo è una proprietà del piano di accesso) ma che le restanti proprietà siano uguali: entrambi i piani sono potenzialmente validi per essere estesi (cioè per essere ulteriormente sviluppati per riuscire ad arrivare al risultato finale). Dato che le proprietà, ad eccezion fatta per il costo, sono identiche, dal punto di vista "esterno" cioè del nodo padre, esse sono indistinguibili cioè producono gli stessi identici risultati (magari con un ordine diverso, ma abbiamo deciso di ignorarlo). Dato che il costo di AP risulta maggiore di quello di AP' è sicuramente inutile espandere AP poichè in qualsiasi modo io prosegua per la risoluzione ulteriore del piano di accesso, il costo non sarà mai inferiore al miglior modo di sviluppare il piano di accesso che partiva però da AP' proprio perchè a costo inferiore: di conseguenza il piano AP può essere ignorato. In caso considerassimo ora anche l'ordine O, se AP e AP' hanno lo stesso ordine allora si ricade nel caso precedente, se l'ordine O non risulta significativo (cioè non traggo alcun vantaggio ad avere un ordinamento piuttosto che un altro) posso ignorarlo e ricadere nuovamente nel caso precedente, se invece hanno ordini diversi ma AP ha un ordine meno significativo (che solitamente vuol dire disordinato) rispetto all'ordine di AP' allora possiamo ancora dire che AP' domina AP (sempre a parità delle restanti proprietà e con il costo di AP' minore di quello di AP). Se invece AP avente costo maggiore di AP', risulta avere un ordine più significativo (o non comparabile) dell'ordine di AP' allora non posso più dire che AP' domina AP. Facciamo un'esempio di quest'ultima affermazione: mettiamo che nella mia query sia presente un ORDER BY *cantina*

e che il piano AP' abbia un ordine significativo sull'attributo sid (poichè, ad esempio, riesce a risolvermi in fretta un join successivo) mentre AP abbia un ordine significativo su *cantina* (poichè risolve completamente l'ORDER BY): i due ordini non risultano comparabili poichè non posso dire a priori quale dei due risulterà migliore, di conseguenza AP' non domina AP e AP non domina AP'.

Facciamo un esempio che risolva la seguente query

```
SELECT R.A, R.B, S.C
FROM R, S
WHERE R.J=S.H AND R.A>5
ORDER BY R.B
```

confrontando di seguito 3 possibili piani di accesso (Fig. 19)

Figura 19:

AP	224
Cost	345
Cardinality	25
Schema	{R.A,R.B,S.C}
Tables	{R,S}
Predicates	R.J=S.J, R.A > 5
Order	R.B

AP	546
Cost	216
Cardinality	25
Schema	{R.A,R.B,S.C}
Tables	{R,S}
Predicates	R.J=S.J, R.A > 5
Order	none

AP	1127
Cost	234
Cardinality	25
Schema	{R.A,R.B,S.C}
Tables	{R,S}
Predicates	R.J=S.J, R.A > 5
Order	R.B

Possiamo dire che AP1127 *domina* AP224 poichè a parità di proprietà, il costo di AP1127 risulta essere minore rispetto a quello di AP224. Allo stesso modo, però, possiamo dire che AP546 non domina AP1127: nonostante abbia un costo inferiore AP546 non risulta ordinato per R.B (cosa che AP1127 è). Si noti che neppure AP1127 domina AP546, poichè nonostante abbia un ordinamento significativo, il suo costo risulta maggiore rispetto a AP546.

Un ulteriore esempio sul *pruning* chirificatore può essere il seguente: si ipotizzi che venga utilizzato solo il *nested loops* per risolvere il join e che non ci sia alcun ordinamento *significativo* (quindi non vi è alcun ORDER BY, GROUP BY, DISTINCT, ...). Si ipotizzi inoltre che per una query effettuata sulle relazioni R1, R2 ed R3 sia sempre più conveniente avere la relazione R1 come esterna; allora con le seguenti ipotesi nessun piano con la sequenza $(R2 \triangleright\triangleleft R1) \triangleright\triangleleft R3$ potrà essere ottimale poichè R1 risulta essere relazione interna.

1.6 Generazione di piani di accesso

Per la generazione di piani di accesso (PdA da ora in poi) si può pensare di utilizzare il seguente **algoritmo di programmazione dinamica:**

- **Livello 1**: determina per ogni R_i e ogni ordine *significativo* O (incluso il caso non ordinato) il piano parziale migliore (al livello 1 implica come accedere ai dati della relazione R)
- **Livello i ($2 \leq i \leq N$)**: Per ogni ordine *significativo* O e per ogni sottoinsieme di *i relazioni* determina il piano parziale migliore *a partire dal risultato del passo i-1*
- **Livello N+1**: determina il piano di accesso ottimale, aggiungendo se necessario il costo di ordinamento finale del risultato

Con questo algoritmo il numero di PdA valutati è $O(2^N)$. L'algoritmo appena descritto opera in modo *breadth-first* (cioè per "livelli" che equivale al numero di relazioni considerate, cioè si noti che ad esempio al livello 2 vengono generati tutti i livelli "promettenti" di due relazioni utilizzando come metodi di accesso quelli provenienti dal passo 1). Nella pratica la generazione di nuovi piani procede in maniera più efficace tipicamente espandendo i piani di accesso parziali più "promettenti" (che tipicamente vuol dire quelli a costo minore) e quindi in maniera *depth-first*. Questo normalmente permette di eliminare un maggior numero di piani parziali poichè permette di raggiungere più velocemente una soluzione "completa" (poichè viene espanso un solo piano per livello) il che potrebbe permette la conclusione celere della generazione del PdA ottimale. A conseguenza di ciò bisogna estendere il concetto di dominazione: se AP' *domina* AP non solo deve avere costo minore per lo stesso ordine significativo O sulle stesse relazioni, ma deve verificare che le relazioni elaborate da AP' includano quelle elaborate da AP. Si passa dunque da *eguaglianza* $\Rightarrow$ *inclusione*.

Esempio 1 Facciamo un esempio: si consideri la seguente query

```
SELECT S.snome, R.rivista
FROM Recensioni as R, Sommelier as S
WHERE R.sid=S.sid AND R.vid=100
        AND S.val=4
```

Sono disponibili indici B+tree *unclustered* (non ordinati) su R.vid, R.sid, S.val e S.sid (un indice per ogni attributo). Gli unici algoritmi di join disponibili sono il *nested loop* e l'*index nested loop*. Le relazioni hanno i seguenti dati

- N(R)=2000, P(R)=400, NK(R.vid)=100, L(R.vid)=20, NK(R.sid)=200, L(R.sid)=25
- N(S)=200, P(S)=100, NK(S.val)=20, L(S.val)=3, NK(S.sid)=200, L(S.sid)=5

Vediamo come l'algoritmo di generazione di PdA procede (i PdA evidenziati in grassetto sono "dominanti").

- **Passo 1**: si valuta il metodo di accesso di costo minimo per ogni relazione. Non vi è alcun ordine significativo da tener conto (poichè non vi è alcun ORDER BY, GROUP BY, ... e perchè l'unico algoritmo di join disponibile è il nested loop).

- Relazione R:
  - Cardinalità risultato= $\frac{2000}{100} = 20$.
  - (AP1) Costo sequenziale= 400
  - **(AP2)** Costo indice R.vid= $\frac{20}{200} + \Phi(\frac{2000}{100}, 400) = 1 + 20 = 21$

  Si vede che AP2 domina AP1
- Relazione S:
  - Cardinalità risultato= $\frac{200}{20} = 10$
  - (AP3) Costo sequenziale= 100
  - **(AP4)** Costo indice S.val= $\frac{3}{20} + \Phi\left(\frac{200}{20}, 100\right) = 1 + 10 = 11$

  Si vede che AP4 domina AP3

- **Passo 2**: si espande AP4 poichè ha costo minimo (costo 11, ricordiamo che stiamo facendo una depth-first quindi si espande per primo sempre il piano con costo minimo) considerando il join che si ha con la relazione R
  - Nested loop:
    - Costo di partenza= 11
    - Numeri di loop= 10
    - Costo per ogni loop= 21 (dato dall'indice su R.vid, si veda AP2)
    - (AP5) Costo del nested loop= $11 + 10 \cdot 21 = 221$
  - Index nested loop (con indice su R.sid):
    - Costo di partenza= 11
    - Numeri di loop= 10
    - Costo per ogni loop= $\frac{25}{200} + \Phi\left(\frac{2000}{200}, 400\right) = 11$
    - **(AP6)** Costo dell'index nested loop= $11 + 10 \cdot 11 = 121$

    AP6 domina AP5.

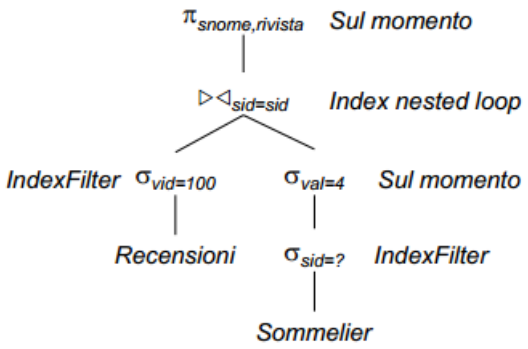
- **Passo 3**: si espande AP2 (con costo 21) poichè al passo 2 abbiamo trovato solo PdA con costi superiori a 21, considerando il join con la relazione S
  - Nested loop:
    - Costo di partenza= 21
    - Numeri di loop= 20
    - Costo per ogni loop= 11 (dato dall'indice S.val, si veda AP4)
    - (AP7) Costo del nested loop= $21 + 20 \cdot 11 = 241$
  - Index nested loop (con indice su S.sid):
    - Costo di partenza= 21
    - Numeri di loop= 20
    - Costo per ogni loop= 1+1 = 2 (poichè sid è chiave della relazione S)

∗ **(AP8)** Costo del nested loop= $21 + 20 \cdot 2 = 61$

AP8 domina AP7.

- **Passo 4**: si cerca di espandere AP8 che ha costo minimo (costo 61). Si vede che tutti i join sono stati effettuati e risolti. Conclusione: AP8 è il piano di accesso di costo minimo. L'albero che viene generato è il seguente (Fig. 20)

Figura 20:



Esempio 2 Prendiamo la seguente query

```
SELECT S.snome, R.rivista, V.cantina
FROM Recensioni as R, Sommelier as S, Vini as V
WHERE R.sid=S.sid AND R.vid=V.vid
        AND S.val=4 AND V.vnome='Merlot'
```

Esistono indici B+tree *unclustered* su R.sid e su R.vid. Esistono indinci B+tree *clustered* su S.sid e V.vid. Si ignorano i costi di accesso ali indici. I dati a disposizione sono:

- N(R)=5000, P(R)=1000, NK(R.sid)=200, NK(R.vid)=100
- N(S)=200, P(S)=100, NK(S.val)=10, NK(S.sid)=200
- N(V)=100, P(V)=20, NK(V.vid)=100, NK(V.vnome)=20

Ordini significativi: *sid* e *vid* (poichè utili in caso di join).

- **Passo 1**: si trovano i metodi di accesso di costo minimo alle relazioni, conservando anche quelli che generano ordinamenti utili
  - (AP1) Scansione di R: Costo= 1000, risultati= 5000, ordine: nessuno
  - (AP2) Indice su R.sid: Costo= 5000, risultati= 5000, ordine: *sid*
  - (AP3) Indice su R.vid: Costo= 5000, risultati= 5000, ordine: *vid*

- (AP4) Scansione di S (con selezione su *val*): Costo= 100, risultati= $\frac{200}{10} = 20$, ordine: *sid*
- (AP5) Scansione di V (con selezione su *vnome*): Costo= 20, risultati= $\frac{100}{20} = 5$, ordine: *vid*

- **Passo 2**: si espande AP5 (Costo minimo: 20) considerando solo il join $V \triangleright\triangleleft R$ (poichè $V \triangleright\triangleleft S$ è un prodotto cartesiano e per l'euristica utilizzata i piani che contengono prodotti cartesiani vengono ignorati)

  - Index nested loop:
    - Costo di partenza= 20
    - Numeri di loop= 5
    - Costo per ogni loop= $\frac{5000}{100} = 50$
    - **(AP6)** Costo= $20 + 5 \cdot 50 = 270$
    - Ordine: *vid*
  - Merge-scan:
    - Costo di partenza= 20
    - Costo di merge= 5000 (poichè indice *unclustered* su R.vid)
    - (AP7) Costo= $20 + 5000 = 5020$
    - Ordine: *vid*

  Dimensione del risultato= $\frac{5000}{20} = 250$ (il 20 al denominatore è dato al numero dei risultati presenti su AP5). AP6 domina AP5.

- **Passo 3**: si espande AP4 (costo 100) poichè al passo 2 sono stati trovati PdA con costi superiori a 100, considerando il solo join $S \triangleright\triangleleft R$

  - Index nested loop:
    - Costo di partenza= 100
    - Numeri di loop= 20
    - Costo per ogni loop= $\frac{5000}{200} = 25$
    - **(AP8)** Costo= $100 + 20 \cdot 25 = 600$
    - Ordine: *sid*
  - Merge-scan:
    - Costo di partenza= 100
    - Costo di merge= 5000 (poichè indice *unclustered* su R.sid)
    - (AP9) Costo= $100 + 5000 = 5100$
    - Ordine: *sid*

  Dimensione del risultato= $\frac{5000}{10} = 500$. AP8 domina AP9.

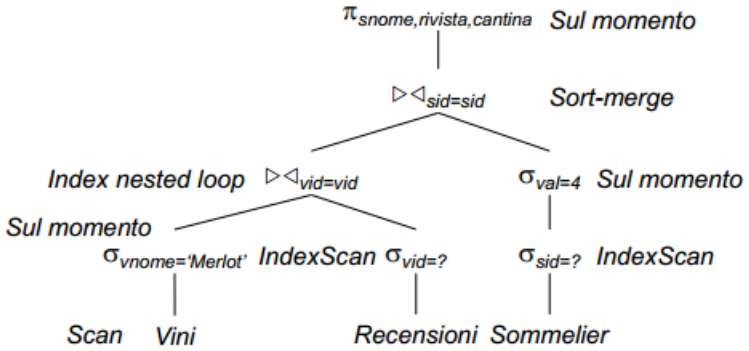
- **Passo 4**: si espande AP6 (costo 270) poichè fra AP6 e AP8 (notiamo che AP7 e AP9 non vengono neppure presi in considerazioni poichè dominati) risulta avere il costo minore, considerando il join con S (la sequenza di join è $(V \triangleright\triangleleft R) \triangleright\triangleleft S$)

  - Index nested loop:
    - Costo di partenza= 270
    - Numeri di loop= 250
    - Costo per ogni loop= 1 (*sid* chiave di S)
    - (AP10) Costo= $270 + 250 \cdot 1 = 520$
  - Sort-merge:
    - Costo di partenza= 270
    - Costo di sort= $4 \cdot P(temp)$ (dove temp è la relazione temporanea per ordinare *sid*)
    - Costo di merge= $P(temp) + P(temp) + 100$
    - **(AP11)** Costo= $270 + 4 \cdot 10 + 10 + 10 + 100 = 430$

  Dimensione del risultato= $\frac{250}{10} = 25$. AP11 domina AP10.

- **Passo 5**: si cerca di espandere AP11 (costo 430). Si vede che tutti i join sono stati effettuati e risolti. Conclusione: AP11 è il piano di accesso di costo minimo in quanti tutti i piani parziali sono dominati da AP11. L'albero che viene generato è il seguente (Fig. 21)

Figura 21:



1.6.1 Ricerca greedy

Talvolta, per diminuire ulteriormente lo spazio di ricerca, si mantiene per ogni livello solamente la soluzione a costo minimo o a parità di costo quella che produce il minor numero di record: il costo della ricerca diventa lieneare nel numero delle relazioni, ma evidentemente può portare a perdere la soluzione migliore.

NB Si voglia ricordare che per query *ad hoc* dobbiamo minimizzare il tempo totale, dato dal costo di trovare il miglior piano di accesso (ottimizzazione) ed effettivamente metterla in pratica (soluzione).