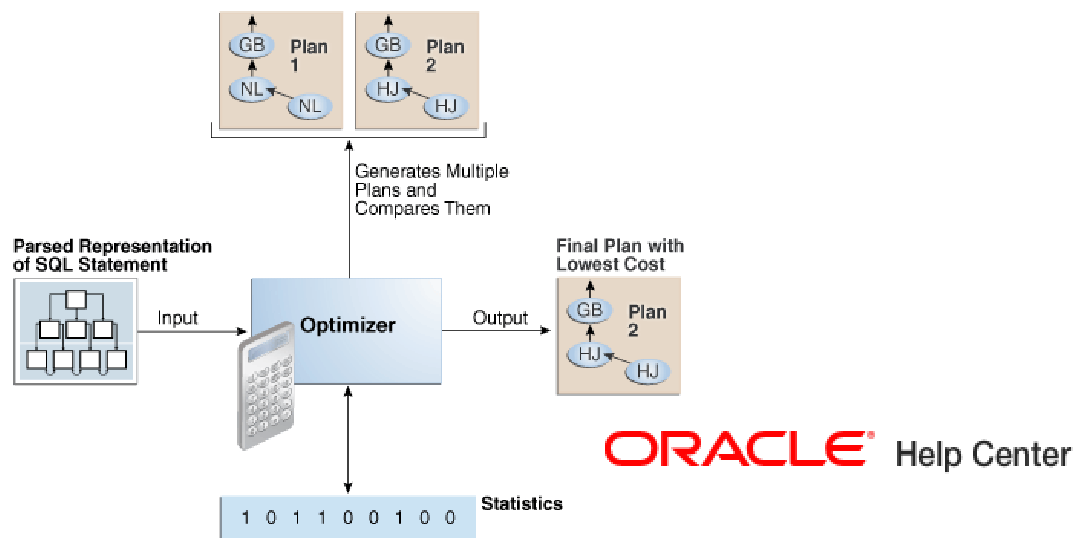
Ottimizzazione di interrogazioni

Tecnologie e Sistemi per la Gestione di Basi di Dati e Big Data M

Ottimizzazione di interrogazioni

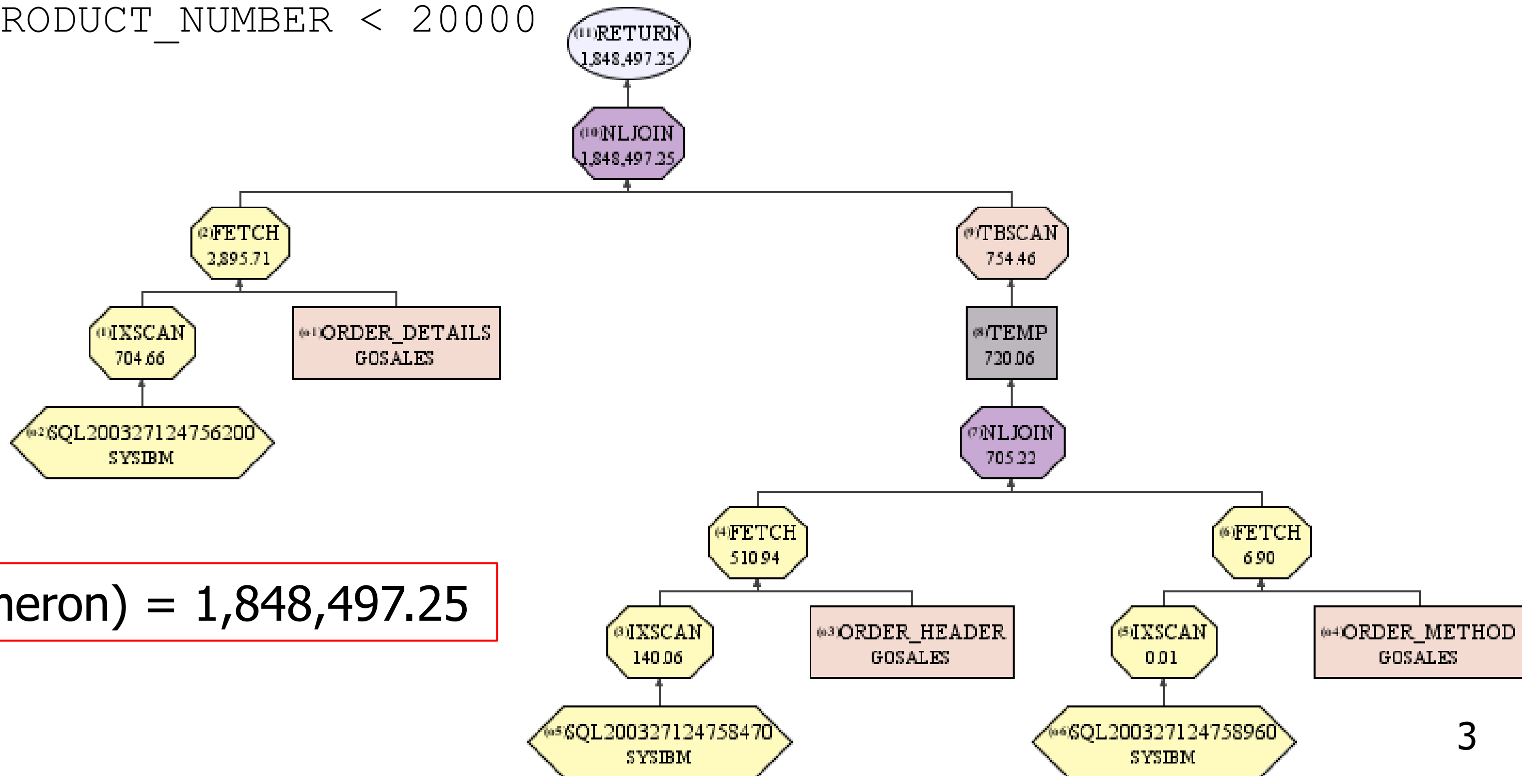
- La risoluzione di un'interrogazione (anche semplice) può essere realizzata in diversi modi, con costi anche molto differenti tra loro (ordini di grandezza!)
- Il DBMS, mediante un modulo ottimizzatore, determina il modo migliore per risolvere una query, generando un piano di accesso che viene stimato avere costo minimo
- A partire da una rappresentazione interna della query, e avvalendosi delle necessarie statistiche, l'ottimizzatore genera una serie di piani di accesso alternativi e tra questi determina il migliore



Ordini di grandezza…

- La query da ottimizzare è

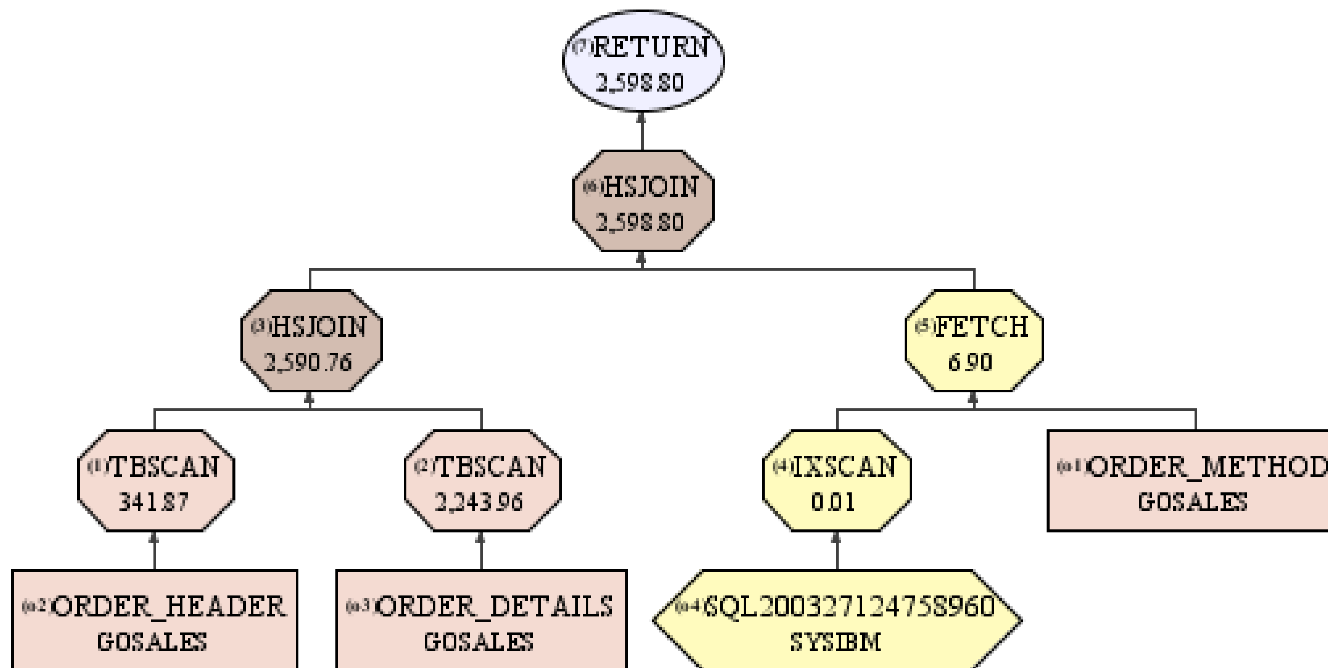
```
SELECT  *
FROM    ORDER_DETAILS OD,                               -- 446023 tuples
        ORDER_HEADER OH,                                -- 53267 tuples
        ORDER_METHOD OM                                 -- 7 tuples
WHERE   OD.ORDER_NUMBER = OH.ORDER_NUMBER
AND     OM.ORDER_METHOD_CODE = OH.ORDER_METHOD_CODE
AND     OD.PRODUCT_NUMBER < 20000
```



Costo (timeron) = 1,848,497.25

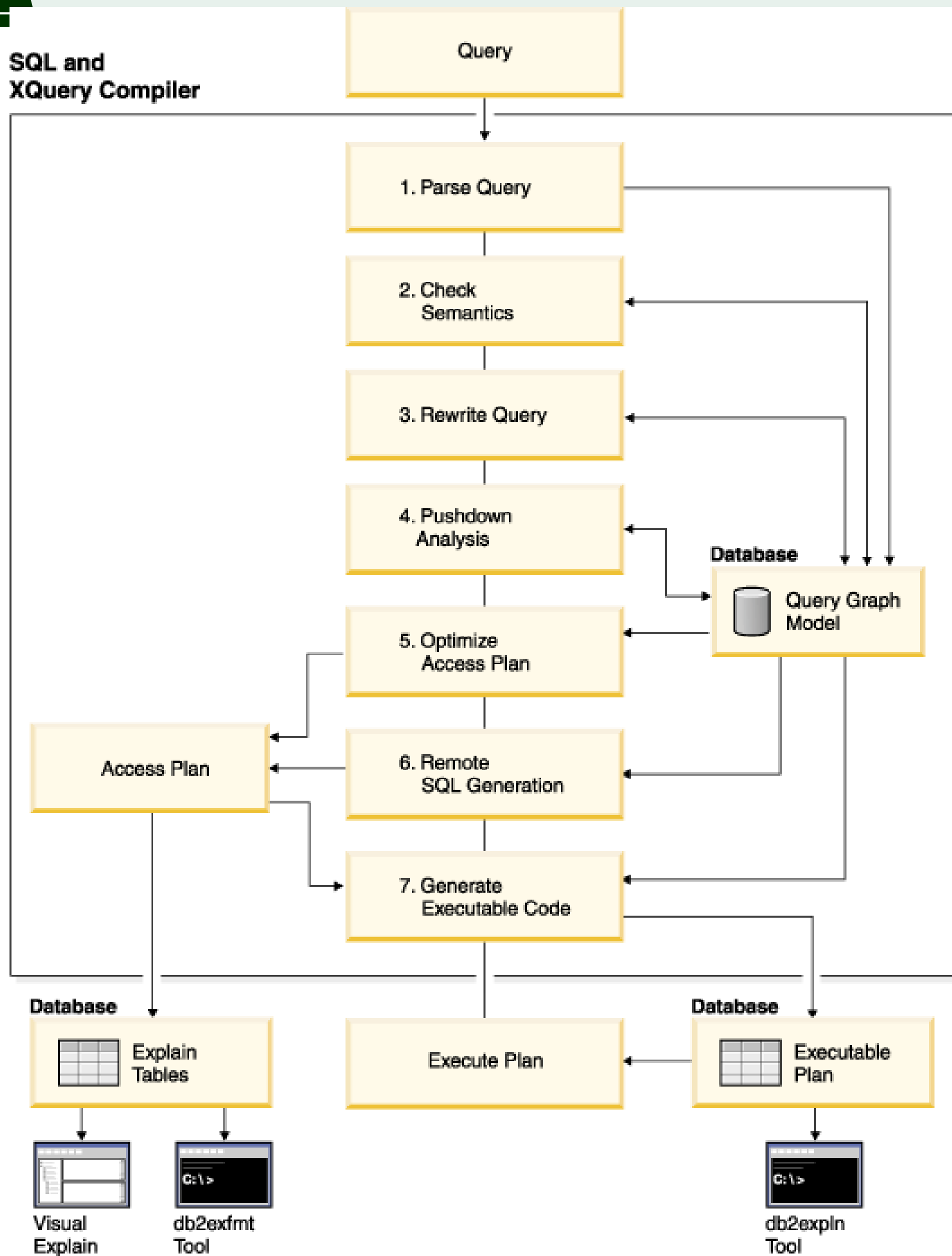
… in meno

- Cambiando tipo di join e ordine delle relazioni…



Costo (timeron) = 2,598.80

Il query compiler di DB2



- Passi salienti:
  - Parse query
  - Check semantics
  - Rewrite query
  - Optimize access plan
  - Generate executable code
- Il tutto si basa su un modello di query a grafo
  - Nodi = operatori
  - Archi = flusso di dati

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, verifica che:
  - Gli oggetti referenziati esistano
  - Gli operatori siano applicati a dati di tipo opportuno
  - L'utente abbia i privilegi necessari per eseguire l'operazione

Check semantici e uso dei cataloghi (i)

- Facciamo riferimento ai cataloghi di DB2 nello schema SYSCAT:

SQL Catalog	SQL attribute	Descrizione
TABLES	TABSCHEMA	Nome dello schema
TABLES	TABNAME	Nome della table, vista o alias
TABLES	DEFINER	Userid del creatore
TABLES	TYPE	T = Table; V = View; A = Alias
COLUMNS	TABSCHEMA	Nome dello schema
COLUMNS	TABNAME	Nome della table o vista
COLUMNS	COLNAME	Nome dell’attributo
VIEWS	VIEWSCHEMA	Nome dello schema
VIEWS	VIEWNAME	Nome della view
VIEWS	DEFINER	Userid del creatore
VIEWS	TEXT	Definizione SQL della vista

Check semantici e uso dei cataloghi (ii)

- … e supponiamo di avere la seguente query:

```
SELECT EmpNo
FROM   MySchema.Employee
```

- In fase di check semantico vengono eseguite query del tipo:

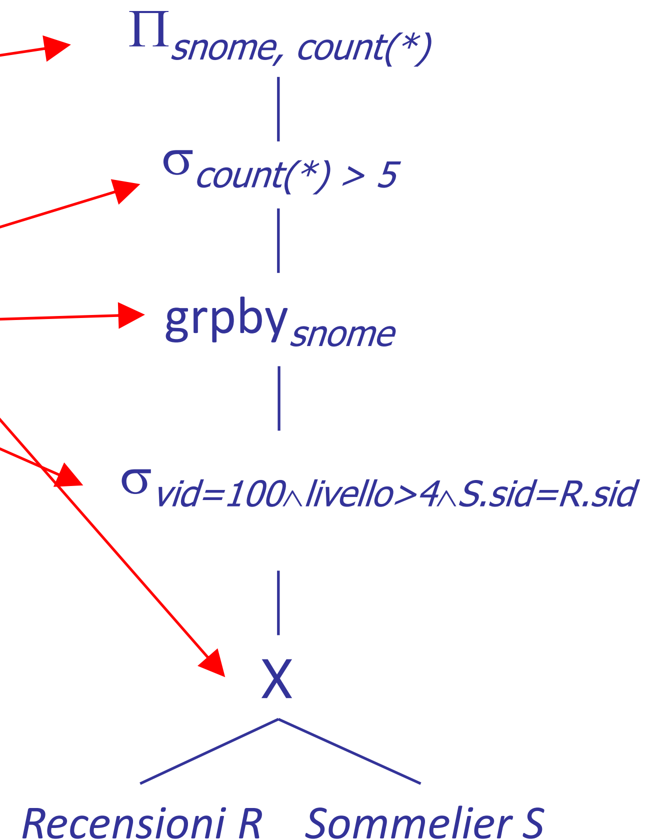
```
SELECT  *
FROM   SYSCAT.TABLES
WHERE  TABSCHEMA = ‘MySchema’
AND    TABNAME = ‘Employee’ ;

SELECT *
FROM   SYSCAT.COLUMNS
WHERE  TABSCHEMA = ‘MySchema’
AND    TABNAME = ‘Employee’
AND    COLNAME = ‘EmpNo’ ;
```

Rappresentazione interna di una query

- Il punto di partenza per il processo di ottimizzazione è dato da una rappresentazione della query ad albero che ricorda da vicino la forma SQL:
  - Foglie = relazioni (o viste o risultati di subquery)
  - Nodi interni = operatori **logici**

```
SELECT S.snome, COUNT(*)
FROM   Recensioni R, Sommelier S
WHERE  R.sid=S.sid
AND    R.vid=100 AND S.livello>4
GROUP BY S.snome
HAVING COUNT(*) > 5
```



Riscrittura di query (query rewriting)

- Prima di procedere alla fase vera e propria di ottimizzazione della query, il DBMS esegue un passo di "rewriting" della stessa
- Lo scopo della fase di riscrittura è semplificare la query e pervenire a una forma più semplice da analizzare e quindi ottimizzare
- Questa fase è talvolta anche detta di ottimizzazione semantica, perché si basa solo sulle proprietà logiche dei dati ed esula da considerazioni di carattere fisico (costi)
- Tra le operazioni tipiche che hanno luogo in questa fase vi sono:
  - Risoluzione delle viste: si esegue il merge della query in input con le query che definiscono le viste referenziate
  - Unnesting: se la query include delle subquery si prova a trasformarla in una forma senza innestamenti
  - Uso dei vincoli: vengono sfruttati i vincoli definiti sugli schemi al fine di semplificare la query
- Il modo con cui vengono eseguite queste operazioni varia da sistema a sistema
- Inoltre il modo cambia anche per uno stesso DBMS se si scelgono "livelli di ottimizzazione" differenti!

Risoluzione di viste

- Per un semplice esempio, si consideri la vista:

```
CREATE VIEW EmpSalaries(EmpNo,Last,First,Salary)
AS   SELECT   EmpNo,LastName,FirstName,Salary
     FROM     Employee
     WHERE    Salary > 20000
```

  e la query:

```
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 > 20000
AND    LastName LIKE 'B%'
```

- Se si avesse un indice su LastName la query riscritta ora lo potrebbe sfruttare

Query innestate

- Le subquery sono blocchi `SELECT-FROM-WHERE` usati nella clausola `WHERE` (o `HAVING`) di un blocco più esterno
- 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 le subquery come se fossero query a parte:
  - Per ogni blocco `SELECT-FROM-WHERE` si produce un piano di accesso
  - Per evitare però di perdere dei piani di accesso a basso costo, l'ottimizzatore cerca di riscrivere la query senza usare subquery

Unnesting senza correlazione: esempio 1

- Il passaggio a una forma senza subquery alle volte è immediato; ad esempio:

```
SELECT   EmpNo, PhoneNo
FROM     Employee
WHERE    LastName LIKE 'L%'
AND      WorkDept IN (SELECT DeptNo
                      FROM   Department
                      WHERE  DeptName = 'Operations' )
```

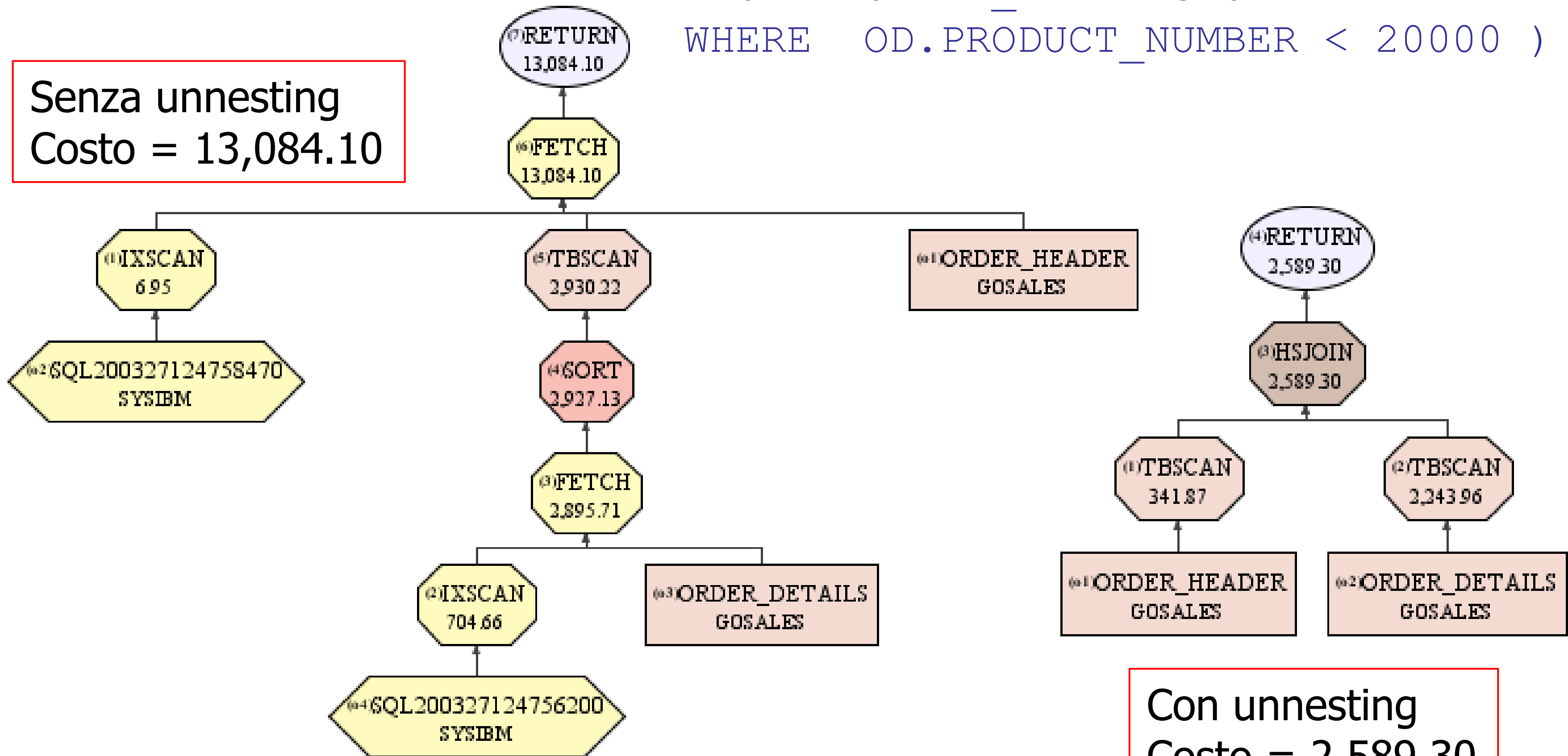
viene riscritta come:

```
SELECT   E.EmpNo, E.PhoneNo
FROM     Employee E, Department D
WHERE    E.WorkDept = D.DeptNo
AND      LastName LIKE 'L%'
AND      D.DeptName = 'Operations'
```

rendendo quindi esplicito il join e fornendo maggior flessibilità all'ottimizzatore

Unnesting senza correlazione: esempio 2

```
SELECT *
FROM   ORDER_HEADER OH
WHERE OH.ORDER_NUMBER IN ( SELECT OD.ORDER_NUMBER
                           FROM   ORDER_DETAILS OD
                           WHERE  OD.PRODUCT_NUMBER < 20000 )
```



Unnesting con correlazione: esempio

- La query:

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

viene riscritta usando un outer join:

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

Uso dei vincoli (i)

- Il DBMS ragiona sfruttando la presenza di vincoli per semplificare le query

- Ad esempio, se `EmpNo` è dichiarata come chiave:

```
SELECT DISTINCT  EmpNo
FROM   Employee
```

la query si riscrive più semplicemente come:

```
SELECT  EmpNo
FROM    Employee
```

Uso dei vincoli (ii)

- Per un esempio di semplificazione basata sul vincolo di Foreign Key, si consideri:

```
SELECT    EA.EmpNo          -- EA.EmpNo è FK
FROM      Emp_Act EA
WHERE     EA.EmpNo IN (SELECT EmpNo FROM Employee)
```

  che viene riscritta come:

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

- Se `EA.EmpNo` ammettesse valori nulli, la riscrittura aggiungerebbe

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

- Benché in molte situazioni può essere l'utente stesso a sfruttare la conoscenza dei vincoli per scrivere una query in forma semplificata, ciò non è sempre possibile…

Uso dei vincoli (iii)

- Si supponga di avere la vista:

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

  e la query:

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

  in cui chi esegue la query non ha privilegio di `SELECT` né su `Employee` né su `Department`

- Il DBMS può però riscrivere la query come:

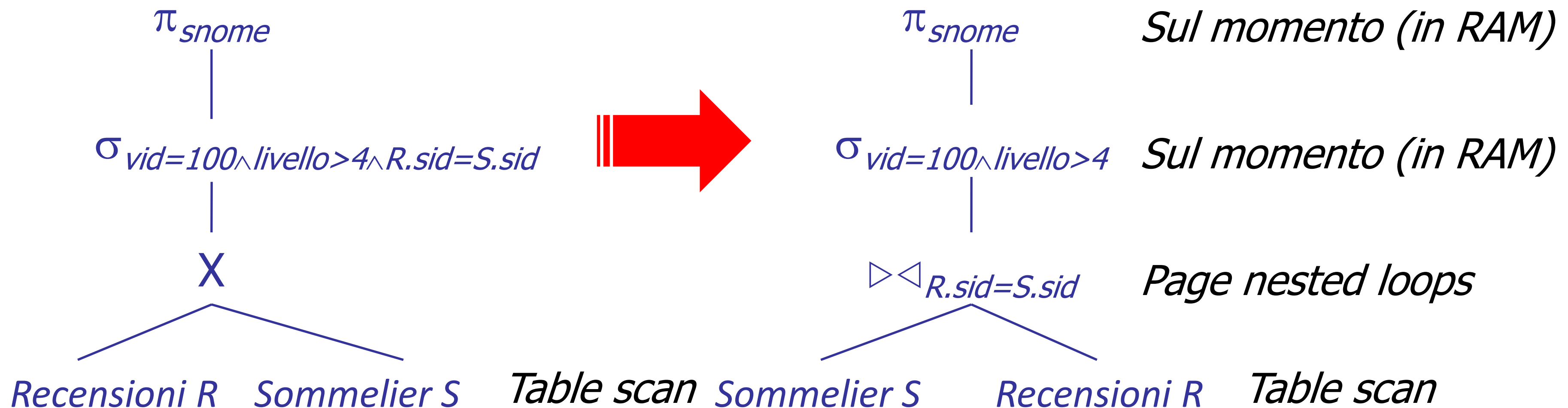
```
SELECT  LastName,FirstName FROM Employee
```

  eventualmente aggiungendo, se `WorkDept` ammette valori nulli:

```
WHERE WorkDept IS NOT NULL
```

Cost-based optimization

- Quella che viene propriamente detta ottimizzazione è la fase che, a partire da una forma interna riscritta, genera un piano di accesso ottimale, ovvero un albero di operatori **fisici** (includendo anche i metodi di accesso alle relazioni)

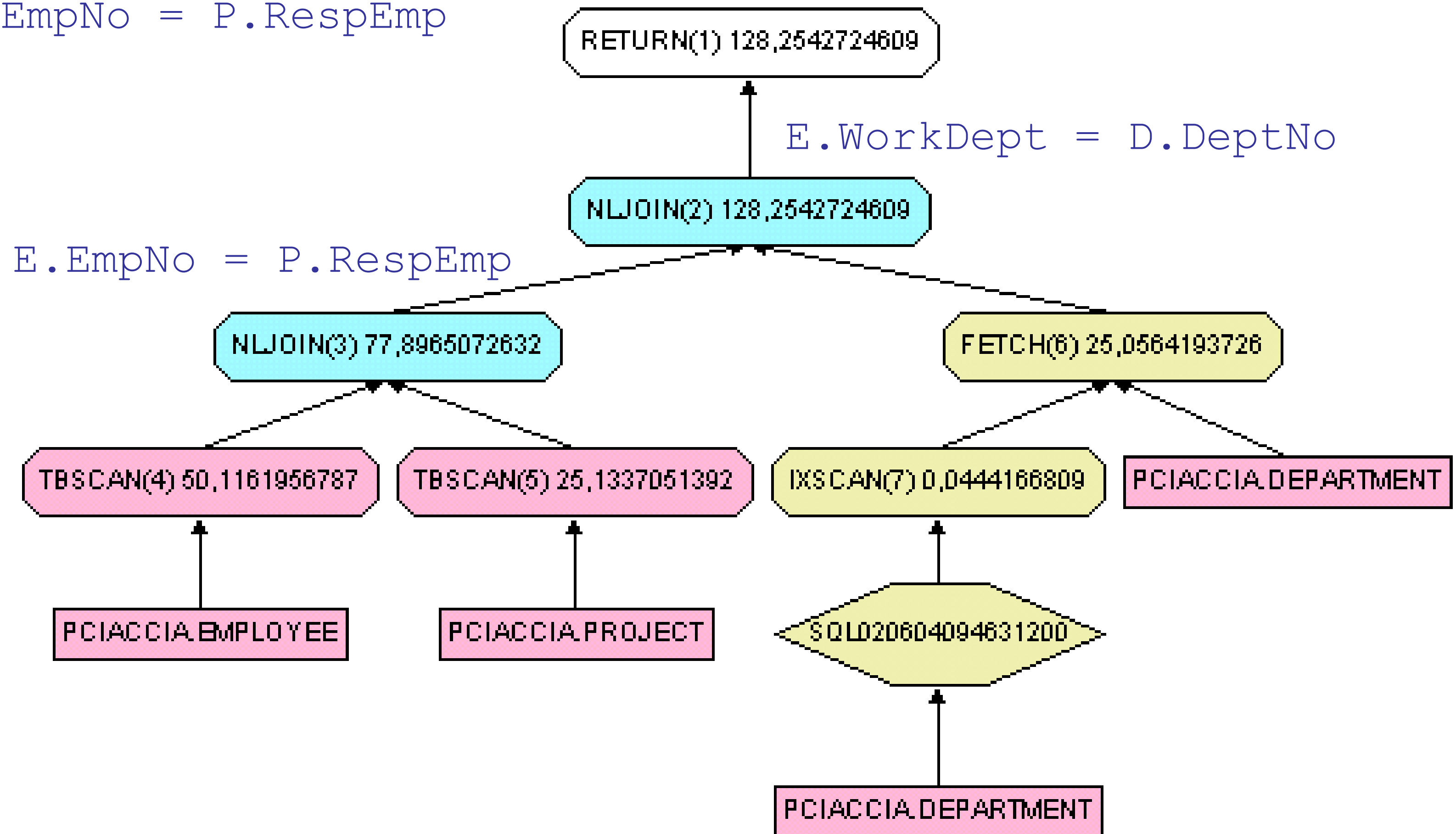


Esecuzione di un piano di accesso

- Per eseguire un piano di accesso occorre valutare gli operatori dell'albero a partire dal basso
- Esistono due possibilità di esecuzione
  - Per materializzazione
    - Ogni operatore memorizza il proprio risultato in una tabella temporanea
    - Un operatore deve attendere che tutti gli operatori di input da cui dipende abbiano terminato l'esecuzione per iniziare a produrre il proprio risultato
  - In pipeline (tramite iteratori)
    - Ogni operatore richiede un risultato agli operatori di input (demand-driven evaluation)
    - Non è "bloccante"
    - Non sempre è possibile (es. sort)
- La valutazione per materializzazione è altamente inefficiente, in quanto comporta la creazione, scrittura e lettura di molte relazioni temporanee, relazioni che, se la dimensione dei risultati intermedi eccede lo spazio disponibile in memoria centrale, devono essere gestite su disco

Esecuzione per materializzazione (i)

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



Esecuzione per materializzazione (ii)

- L'esecuzione per materializzazione produrrebbe come risultato del primo Join:

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

A partire da tale risultato intermedio si può poi calcolare il secondo join

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

- Un modo alternativo di eseguire un piano di accesso è quello di eseguire più operatori in pipeline, ovvero non aspettare che termini l'esecuzione di un operatore per iniziare l'esecuzione di un altro
- Nell'esempio visto, si inizia a eseguire il primo join `E.EmpNo=P.RespEmp`
- Appena viene prodotta la prima tupla:

PROJNO	EMPNO	WORKDEPT
AD3100	000010	A00

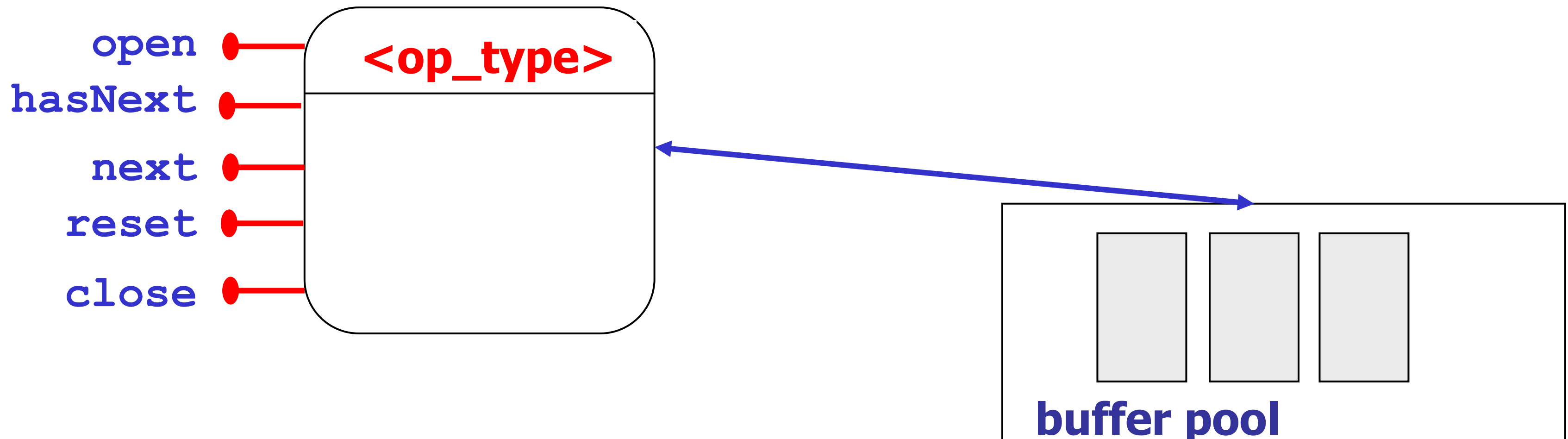
questa viene passata in input al secondo join, `E.WorkDept=D.DeptNo`, che può quindi iniziare la ricerca di matching tuples e quindi produrre la prima tupla del risultato finale della query:

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

- L'esecuzione prosegue cercando eventuali altri match per la tupla prodotta dal primo join; quando è terminata la scansione della relazione interna (`Department`), il secondo join richiede al primo di produrre un'altra tupla

Operatori: interfaccia a iteratore

- Per semplificare il codice di coordinamento dell'esecuzione dei piani di accesso si usa un'interfaccia standardizzata per gli operatori:
- Metodi:
  - open: inizializza, alloca buffer, passa parametri e richiama ricorsivamente open sui figli
  - hasNext: verifica se ci sono altre tuple
  - next: richiede la prossima tupla
  - reset: riparte dalla prima tupla (es. nested loops)
  - close: termina e rilascia le risorse



Pipeline con iteratori

- L'interfaccia a iteratore supporta naturalmente un'esecuzione in pipeline degli operatori, in quanto la decisione se lavorare in pipeline o materializzare è incapsulata nel codice specifico dell'operatore
  - Se l'algoritmo dell'operatore permette di elaborare completamente una tupla in input appena questa viene ricevuta, allora l'input non viene materializzato e si può procedere in pipeline
    - E' questo il caso del Nested Loops Join
  - Se, viceversa, la logica dell'algoritmo richiede di esaminare la stessa tupla in input più volte, allora si rende necessario materializzare
    - E' questo il caso del Sort, che non può produrre in output una tupla senza prima aver esaminato tutte le altre

- E' importante osservare che l'interfaccia a iteratore viene usata anche per incapsulare i metodi di accesso quali i B^+-tree, che esternamente vengono quindi visti semplicemente come operatori che producono un insieme (stream) di tuple

Ricerca del piano di accesso ottimale

- La ricerca del miglior piano di accesso avviene enumerando i possibili piani all'interno di un prefissato spazio di ricerca
  - Spazio di ricerca: tutti i possibili piani, ma non necessariamente
  - Cercando, come vedremo, di non considerare quelli provatamente non ottimali
- Per definizione tutti i piani considerati sono equivalenti dal punto di vista logico, e quindi sono trasformabili l'uno nell'altro sfruttando le regole di equivalenza degli operatori (con NULL e duplicati), ad esempio:
  - Raggruppamento e commutatività della selezione
  - Commutatività e associatività del join
  - Push-down della selezione sul join
- E' evidente che per ogni piano di accesso è necessario stimare il suo costo
  - A partire da quello di esecuzione dei singoli operatori, estrapolando informazioni sulla cardinalità dei risultati prodotti da ciascuno di questi
- Iniziamo a vedere un po' di dettagli relativi all'ultimo punto…

Esempio

- Supponiamo di avere la query:

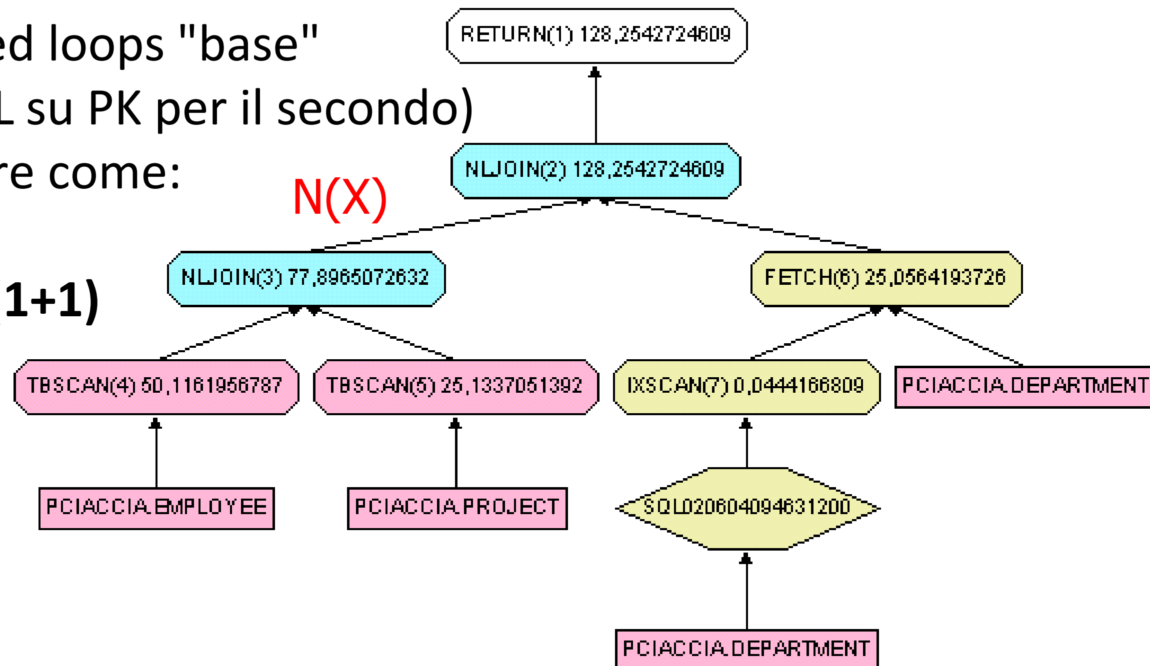
```
SELECT     p.ProjNo, e.EmpNo, d.*
FROM       Department d, Employee e,  Project p
WHERE      e.WorkDept = d.DeptNo
AND        e.EmpNo = p.RespEmp
AND        e.LastName LIKE 'B%'         -- selettività = f
```

  e di considerare il seguente piano di accesso

- Supponendo di usare nested loops "base" per il primo join (e index NL su PK per il secondo) il costo di I/O si può valutare come:

P(e) + f*N(e)*P(p) + N(X)*(1+1)

in cui N(X) è il numero di tuple prodotte dal primo join



Database profile

- L'approccio più diffuso nei DBMS consiste nel mantenere nei cataloghi del DB informazioni statistiche su table, indici e attributi (DB profile)
- A partire da queste, applicando formule/tecniche varie si possono poi stimare le cardinalità dei risultati intermedi prodotti dai vari operatori
- In genere si fanno ipotesi semplificative per rendere il problema gestibile e/o perché mancano informazioni più dettagliate

- Approccio alternativo: ottenere informazioni run-time eseguendo la query su un piccolo campione dei dati in input, e quindi estrapolando al caso completo
  - Compromesso tra dimensione del campione e accuratezza delle stime estrapolate

IBM DB2

GSDB: GoSales DB

RETURNED_ITEM
123 RETURN_CODE
123 ORDER_DETAIL_CODE
123 RETURN_REASON_CODE

RETURN_REASON
123 RETURN_REASON_CODE

SALES_TARGET
123 SALES_YEAR
123 SALES_PERIOD
123 COUNTRY_CODE_RETAILER
123 SALES_STAFF_CODE
123 RETAILER_CODE
123 PRODUCT_TYPE_CODE
123 PRODUCT_BRAND_CODE

PRODUCT_TYPE
123 PRODUCT_TYPE_CODE
123 PRODUCT_LINE_CODE

PRODUCT_BRAND
123 PRODUCT_BRAND_CODE

PRODUCT_NAME_LOOKUP
123 PRODUCT_NUMBER
ABC PRODUCT_LANGUAGE

ORDER_DETAILS
123 ORDER_DETAIL_CODE
123 ORDER_NUMBER
123 PRODUCT_NUMBER

PRODUCT
123 PRODUCT_NUMBER
123 PRODUCT_TYPE_CODE
123 PRODUCT_COLOR_CODE
123 PRODUCT_SIZE_CODE
123 PRODUCT_BRAND_CODE

PRODUCT_COLOR_LOOKUP
123 PRODUCT_COLOR_CODE

INVENTORY_LEVELS
123 INVENTORY_YEAR
123 INVENTORY_MONTH
123 WAREHOUSE_BRANCH_CODE
123 PRODUCT_NUMBER

PRODUCT_SIZE_LOOKUP
123 PRODUCT_SIZE_CODE

XGOREV
123 GOREV_ID

ORDER_METHOD
123 ORDER_METHOD_CODE

ORDER_HEADER
123 ORDER_NUMBER
123 RETAILER_SITE_CODE
123 RETAILER_CONTACT_CODE
123 SALES_BRANCH_CODE
123 ORDER_METHOD_CODE

BRANCH
123 BRANCH_CODE
123 COUNTRY_CODE

COUNTRY
123 COUNTRY_CODE
123 SALES_REGION_CODE

TIME_QUARTER_LOOKUP
123 QUARTER_KEY

TIME_DIMENSION
123 DAY_KEY
123 QUARTER_KEY

PRODUCT_FORECAST
123 SALES_YEAR
123 SALES_MONTH
123 BRANCH_CODE
123 BASE_PRODUCT_NUMBER

CONVERSION_RATE
123 COUNTRY_CODE
123 CONVERSION_YEAR
123 CONVERSION_MONTH

CURRENCY_LOOKUP
123 COUNTRY_CODE

Le statistiche di DB2: tables

- Mantenute all'interno dello schema SYSSTAT

```
SELECT  TABNAME, CARD, NPAGES, FPAGES
FROM    SYSSTAT.TABLES
WHERE   TABSCHEMA = 'GOSALES'
AND     CARD > 100;
```

TABNAME	CARD	NPAGES	FPAGES
CONVERSION_RATE	624	3	4
INVENTORY_LEVELS	53837	212	224
ORDER_DETAILS	446023	2001	2016
ORDER_HEADER	53267	312	320
PRODUCT	274	2	3
PRODUCT_FORECAST	129096	509	512
PRODUCT_NAME_LOOKUP	6302	79	96
RETURNED_ITEM	10249	41	64
SALES_TARGET	233625	1010	1024
TIME_DIMENSION	1465	62	64

FPAGES: Pagine usate da una table (P)
NPAGES: Pagine contenenti record (≤P)
CARD: Record in una table (N)

Le statistiche di DB2: attributi

```
SELECT  COLNAME, COLCARD, LOW2KEY, HIGH2KEY
FROM    SYSSTAT.COLUMNS
WHERE   TABSCHEMA = 'GOSALES'
AND     TABNAME = 'ORDER_DETAILS';
```

COLNAME	COLCARD	LOW2KEY	HIGH2KEY
ORDER_DETAIL_CODE	446023	1000002	8290166
ORDER_NUMBER	53248	100002	834935
PRODUCT_NUMBER	260	2110	154140
PROMOTION_CODE	112	10201	90118
QUANTITY	4672	2	15555
SHIP_DATE	1344	'2004-01-19'	'2007-08-30'
UNIT_COST	1264	0.87	660.65
UNIT_PRICE	190	3.66	1271.08
UNIT_SALE_PRICE	720	1.92	1318.93

COLCARD: Numero di valori distinti (NK)
HIGH2KEY,LOW2KEY: Secondo valore maggiore/minore dell'attributo

Le statistiche di DB2: indici

- Le statistiche relative agli indici sono numerose, e includono anche informazioni dettagliate sul clustering e l'allocazione sequenziale delle foglie

```
Name                             |Value
---------------------------------|---------------------------------------
INDNAME                          |SQL200327124759340
TABSCHEMA                        |GOSALES
TABNAME                          |PRODUCT_FORECAST
COLNAMES                         |+SALES_YEAR+SALES_MONTH+BRANCH_CODE+
                                                    BASE_PRODUCT_NUMBER
NLEAF                            |221
NLEVELS                          |2
FIRSTKEYCARD                     |4
FIRST2KEYCARD                    |43
FIRST3KEYCARD                    |1211
FIRST4KEYCARD                    |129096
FULLKEYCARD                      |129096
```

Fattore di selettività di predicati locali

- Basandosi su un'ipotesi di distribuzione uniforme (dei valori sulle tuple) si ha:
  - A = v: $f = 1/NK(A)$
  - A IN (v1,...,vn): $f = n/NK(A)$
  - A < v: $f = (v - min(A))/(max(A) - min(A))$
  - A between v1 and v2: $f = (v2 - v1)/(max(A) - min(A))$
- In mancanze di statistiche si usano selettività costanti:
  - A = v: $f = 1/10$
  - A < v: $f = 1/2$
  - A between v1 and v2: $f = 1/3$
- Assumendo poi indipendenza dei predicati si ottiene:
  - P and Q: $f = f_P \times f_Q$
  - not P: $f = 1 - f_P$
  - P or Q: $f = f_P + f_Q - f_P \times f_Q$

Selettività con più predicati

- Data la condizione P1 and P2 and … and Pn, e supponendo note solo le selettività dei singoli predicati, DB2 stima la selettività complessiva come:

$$f = f_1 \times f_2 \times \ldots \times f_n$$

- La cosa diventa più complessa se si hanno informazioni sulle selettività combinate di 2 o più predicati
- Ad es. se si conosce la selettività complessiva di P1 and P2, $f_{1,2}$, allora la stima diventa:

$$f = f_{1,2} \times f_3 \times \ldots \times f_n$$

  che è più precisa se P1 e P2 non sono indipendenti
- Nel caso generale il problema è complicato, in quanto non è immediatamente chiaro come usare correttamente tutta l'informazione a disposizione (ad es. che fare se si conosce anche $f_{2,3}$?)
- La soluzione, descritta in [MHK+07], si basa sull'applicazione del principio di massima entropia

Selettività dei join

- Il fattore di selettività di un predicato di join tra 2 relazioni R e S determina la cardinalità del risultato, che si esprime in generale come $f_J * N(R) * N(S)$
- Per join di uguaglianza (R.A = S.B) si assume che ogni valore dell'attributo di join con meno valori distinti trovi un match nell'altro attributo (ipotesi di contenimento), quindi $f_J = 1/\max\{NK(R.A), NK(S.B)\}$
- Esempio: N(R) = 10K, N(S) = 2000; NK(R.A) = 1000, NK(S.B) = 500
  - Il join produce 10K*2000/max{1000,500} = 20K tuple
  - Ragionamento (basato sulle ipotesi di contenimento e uniformità):
    - Ogni valore di R.A è ripetuto 10K/1000=10 volte, e ogni valore di S.B 2000/500=4 volte
    - Ogni valore comune produce quindi 10*4 = 40 tuple
    - Poiché ci sono 500 valori in comune, il risultato ha cardinalità 20K

Join Primary key-Foreign key (PK-FK)

- Se R.A è chiave e S.B foreign key la cardinalità del join è pari a N(S)
  - Coerente con la formula vista, poiché in questo caso è $NK(S.B) \leq NK(R.A)$, $NK(R.A) = N(R)$ e quindi $f_J = 1/N(R)$

Cardinalità delle Proiezione

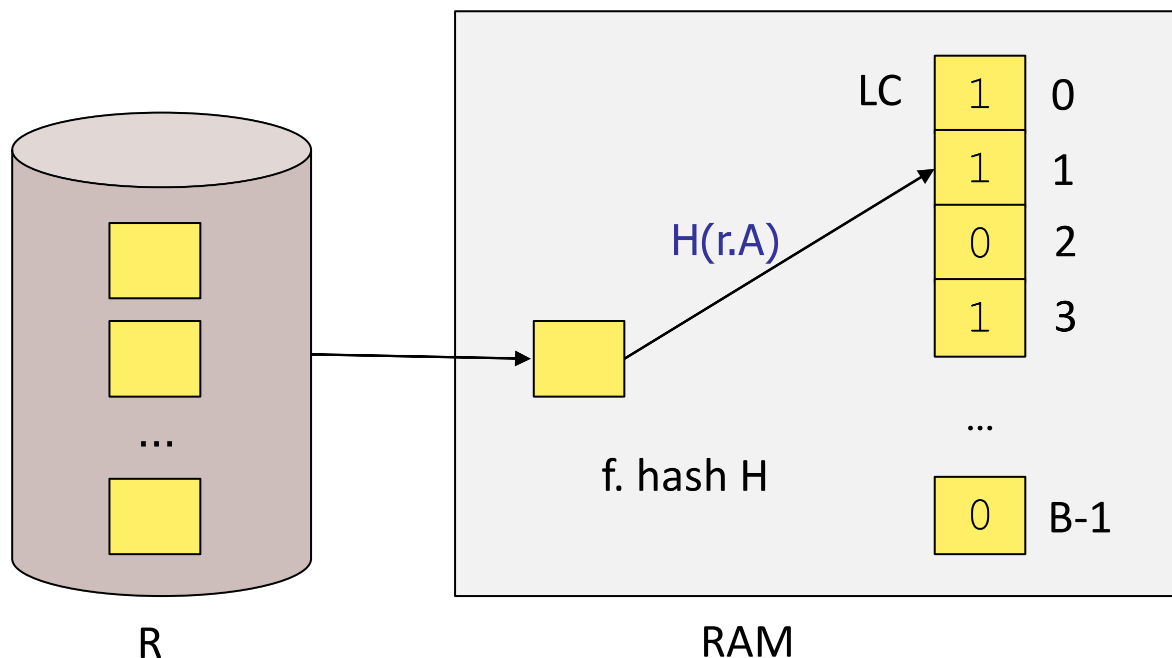
- Se la proiezione di R è su un solo attributo A, il numero di tuple E del risultato è pari a $E = NK(A)$
- Se NK(A) non è noto, si assume il caso peggiore: $E = N(R)$
- Lo stesso si ottiene se la proiezione mantiene una chiave: $E = N(R)$
- Sempre ragionando nel caso peggiore si può stimare il caso di proiezione su più attributi non chiave come: $E = \min\{N(R), \prod_i NK(A_i)\}$

Stima del numero di valori distinti

- Sia il caso in cui NK(R.A) non sia noto, sia quello di proiezione su più attributi non chiave portano a delle stime pessimistiche che possono influenzare negativamente la qualità del piano di accesso selezionato dall'ottimizzatore
- Per ovviare a ciò sono possibili diverse strategie:
  - Collezionare statistiche esatte (costoso)
    - Ad es. un approccio basato su sorting richiede O(P log P) per ogni attributo
  - Usare tecniche approssimate
    - Campionamento
    - (Metodi hash +) modello probabilistico (urn model à la Cardenas)
- Vediamo due esempi di tecniche del secondo tipo…

Linear counting

- Si predispone in RAM un array LC di B bit, inizialmente tutti a 0
- Si scandisce la relazione R e, per ogni tupla r, si applica una funzione hash H a valori in [0,B-1] al valore dell'attributo A di interesse
- Si pone LC[H(r.A)] = 1
- Al termine si conta il numero Z di bit rimasti a 0
- Si stima il numero di valori distinti di A come: NK(R.A) ≈ B ln (B/Z)



Linear counting: analisi

- Basata sullo stesso modello della formula di Cardenas
- Poiché il numero di duplicati di ogni valore non ha influenza su Z, si ha che in media vale la relazione:

$$B - Z = B\,[1-(1-1/B)^{NK}] = B\,[1-(1-1/B)^{B*NK/B}] \approx B - B\,e^{-NK/B}$$

e quindi: $Z \approx B\,e^{-NK/B} \Rightarrow \ln(Z/B) \approx -NK/B$

$$\Rightarrow NK \approx B \ln(B/Z)$$

- Perché il metodo funzioni occorre garantire $Z > 0$
  - Ad es., ponendo $B = N$
- Il metodo è ovviamente applicabile a più attributi, o combinazioni, in parallelo

Cardinalità di proiezioni multi-attributo

- Non volendo fare assunzioni di caso peggiore si può adottare un modello probabilistico basato sulla conoscenza delle cardinalità dei singoli attributi
- Se si proietta una relazione R sugli attributi A e B, si può stimare che il numero medio di tuple nel risultato sia:

$$NK(A)*NK(B)*[1-(1-1/(NK(A)*NK(B)))^{N(R)}]$$

- Il modello probabilistico è quello usato dalla formula di Cardenas e dal Linear Counting, in cui si hanno delle "urne" e si lanciano degli "oggetti", e si contano le urne "colpite" almeno una volta:

Problema	Urne	Oggetti
Pagine che contengono almeno una tupla	P pagine	R tuple selezionate
Valori distinti	B bit	NK valori (incogniti)
Combinazioni di valori distinti	NK(A)*NK(B)	N tuple

Operatori insiemistici

- Per unione e differenza si adotta ancora un approccio basato sul caso peggiore in termini di cardinalità, e quindi
  - N(R) + N(S) per l'unione
  - N(R) per la differenza
- Entrambe le stime in pratica assumono che l'intersezione sia nulla e quindi sono chiaramente in contrasto con la stima che si userebbe se si facesse il join tra le relazioni su tutti gli attributi, basata sull'ipotesi di contenimento
- La stima per l'intersezione sarebbe infatti:

$$N(R)*N(S)/\max\{N(R),N(S)\} = \min\{N(R),N(S)\}$$

Problema: come si può usare l'algoritmo di Linear Counting per stimare la cardinalità di unione, differenza e intersezione?

Istogrammi: motivazione

- Il solo numero di valori distinti può portare a stime poco accurate, che possono quindi influenzare in modo impredicibile il processo di ottimizzazione
- Es.: Dati N dipendenti e NK(ruolo) ruoli:
  - quanti dipendenti hanno il ruolo ‘operaio’? N/NK(ruolo) (?)
  - quanti ‘direttore di filiale’? N/NK(ruolo) (?)
- La distribuzione uniforme è spesso (molto) lontana dalla realtà dei dati
  - Vale anche per domini numerici
- Praticamente tutti i DBMS si basano su istogrammi per ottenere stime più accurate

Istogrammi: concetto generale

- Un istogramma consiste di B intervalli (o "bucket") che partizionano l'insieme dei valori di un attributo A
- Il bucket i-esimo è l'intervallo $(b_{i-1},b_i]$ e memorizza il numero di tuple di R per cui è $b_{i-1} < R.A \leq b_i$
  - I b_i sono anche detti "boundary values"

```
SELECT  SEQNO, COLVALUE, VALCOUNT
FROM    SYSSTAT.COLDIST
WHERE   TABSCHEMA = 'GOSALES'
AND     TABNAME = 'ORDER_DETAILS'
AND     COLNAME = 'QUANTITY'
AND     TYPE = 'Q';
```

VALCOUNT = n. di tuple con valore ≤ COLVALUE

DB2 riporta info sulla distribuzione cumulativa

IBM DB2

SEQNO	COLVALUE	VALCOUNT
1	1	892
2	8	26315
3	14	50178
4	20	72702
5	25	93888
6	31	121764
7	38	143842
8	46	168151
9	56	190006
10	69	212307
...	...	...

Tipi di istogrammi

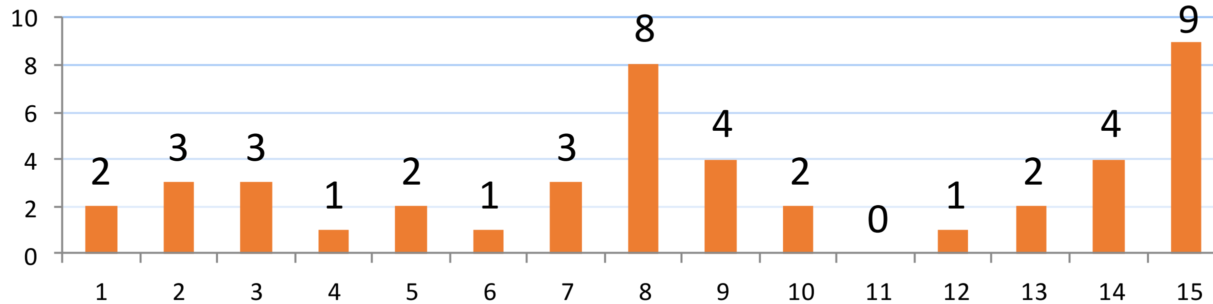
- Equi-width: il dominio viene suddiviso in B intervalli della stessa ampiezza
  - Quindi l'ampiezza (width) w di ogni bucket è costante: $b_i - b_{i-1} = w$
  - Semplici da aggiornare (se gli estremi del dominio non cambiano)
  - Non offrono garanzie sull'errore che si può commettere

- Equi-depth: il dominio viene suddiviso in B intervalli, in modo che il numero di tuple in ogni intervallo sia (circa) lo stesso
  - Più onerosi da aggiornare
  - In grado di adattarsi a distribuzioni non uniformi

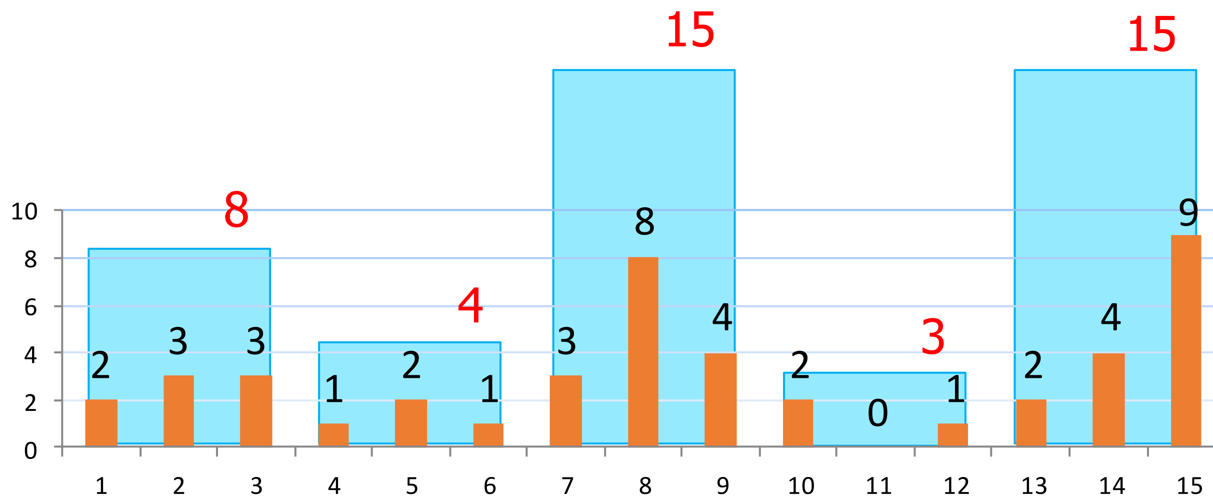
- Gli istogrammi Compressed sono un'estensione degli equi-depth, usati da molti DBMS tra cui DB2, in cui viene mantenuto anche un contatore separato per ognuno dei V valori più frequenti (MCV = Most Common Values)

Istogrammi equi-width

- La distribuzione di N=45 tuple, con NK=15 valori distinti tra 1 e 15, è:

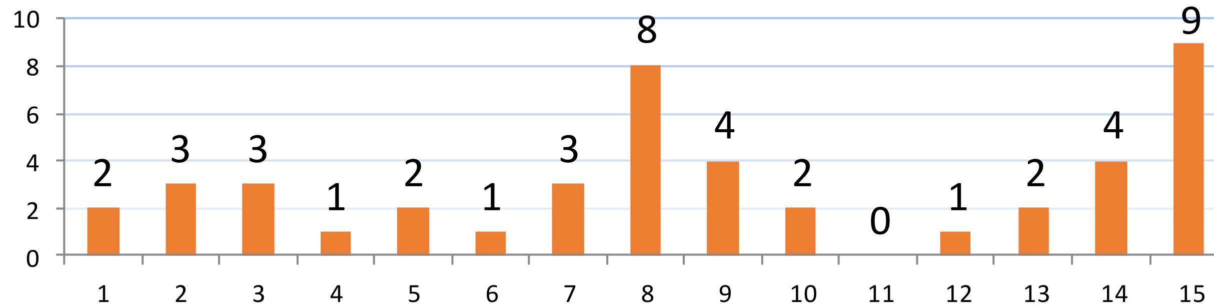


Istogramma Equi-width. Con B=5 è w = 3

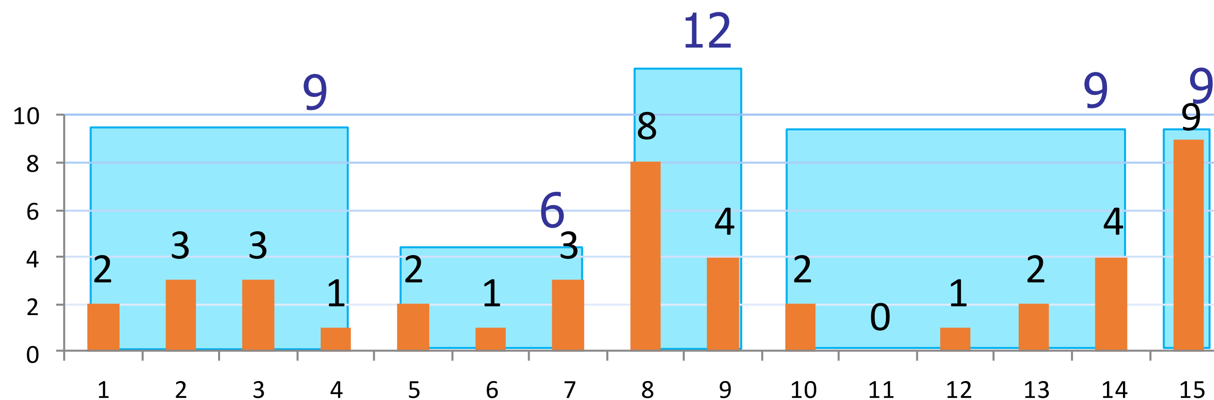


Istogrammi equi-depth

- La distribuzione di N=45 tuple, con NK=15 valori distinti tra 1 e 15, è:



Istogramma Equi-depth. Ogni bucket ha circa 45/5 = 9 tuple



MCV: valori più frequenti

- La conoscenza delle ripetitività dei valori più frequenti, oltre che per stimare meglio la selettività di un predicato, può servire a fornire stime più precise sulla selettività dei join

```
SELECT  SEQNO, COLVALUE, VALCOUNT
FROM    SYSSTAT.COLDIST
WHERE   TABSCHEMA = 'GOSALES'
AND     TABNAME = 'ORDER_DETAILS'
AND     COLNAME = 'QUANTITY'
AND     TYPE = 'F';
```

`TYPE = 'F'`: `VALCOUNT` si riferisce al n. di tuple con valore = `COLVALUE`

IBM DB2

SEQNO	COLVALUE	VALCOUNT
1	14	5798
2	26	5798
3	30	5352
4	24	5352
5	31	4906
6	3	4683
7	29	4460
8	10	4460
9	16	4460
10	6	4237

Cardinalità di relazioni derivate

- La stima della cardinalità di una relazione ottenuta dall'applicazione di più operatori è un problema complesso, che si scontra con l'oggettiva difficoltà di mantenere statistiche multivariate (istogrammi multidimensionali) che possano descrivere le correlazioni esistenti tra diversi attributi
  - Il numero di casi da considerare è esponenziale nel numero di attributi coinvolti
- Le alternative possibili sono essenzialmente due:
  - Fornire stime nel caso peggiore in termini di cardinalità
  - Fornire stime basate su modelli probabilistici
- Vediamo due semplici esempi…

Stima di cardinalità: selezione + proiezione

- Consideriamo la seguente query:

```
SELECT DISTINCT MATRICOLA
FROM   ESAMI
WHERE  VOTO = 31
```

```
N(ESAMI) = 140000

COLNAME              COLCARD (NK)
---------------------------------
MATRICOLA                   14000
COD_CORSO                     700
VOTO                           14
DATA                         1400
```

- La selezione produce N(ESAMI)/NK(VOTO) = 10000 tuple
- Poiché NK(MATRICOLA) > 10000 si stima (caso peggiore) che il risultato finale consista di 10000 tuple
  - Vale a dire: nessuno ha mai preso due o più 30 e lode...
- Viceversa, usando il modello a urne visto, prima si stima che le coppie distinte (MATRICOLA, VOTO) sono 100050
- Poi si divide per NK(VOTO) e si ottiene 7146 come stima del risultato finale
- Equivalentemente: se si selezionano 140000/14=10000 tuple, quanti valori distinti di MATRICOLA ci sono? $\Phi(10000,14000) = 7146$

Join PK-FK con predicati locali

- Dato il join R.A = S.B, con R.A chiave e S.B foreign key, sappiamo che la cardinalità del join è pari a N(S)
- Se è presente anche un predicato locale su S (es. S.C = 15) con selettività f_S il risultato ha cardinalità f_S*N(S)
  - L'ipotesi di contenimento è ancora soddisfatta
- Se è presente un predicato su R (es. R.D = 'abc') con selettività f_R allora il contenimento potrebbe non essere più garantito
- E' il caso della query vista precedentemente (non tutti i progetti hanno come responsabile un dipendente che inizia con B):

```
    e.EmpNo = p.RespEmp   -- join PK-FK
AND e.LastName LIKE 'B%'
```

  - Esiste correlazione tra iniziale del cognome e numero di progetti?
- Se però si prova a usare la formula generale (in cui si pone NK(S.B) = N(R)):

$$1/\max\{f_R*N(R),N(R)\}*(f_R*N(R))*N(S) = f_R*N(S)$$

si ottiene una stima (usata da DB2) che può essere molto lontana dalla realtà

Join PK-FK con predicati locali: esempio 1

- Si considerino le query

```
SELECT *                                -- Q1
FROM   STUDENTI S, ESAMI E
WHERE  E.MATRICOLA = S.MATRICOLA
AND    E.VOTO = 31;                          -- selettività = 0.05
SELECT *                                -- Q2
FROM   STUDENTI S, ESAMI E
WHERE  E.MATRICOLA = S.MATRICOLA
AND    S.REGIONE = 'Emilia Romagna'; -- selettività = 0.2
```

- Q1 produce 0.05*N(ESAMI) tuple
- Viceversa, la stima di 0.2*N(ESAMI) per Q2 è corretta solo se la distribuzione degli studenti sulle varie regioni si mantiene anche per il numero di esami sostenuti
- In questo esempio l'ipotesi NK(S.B) = N(R) equivale a NK(E.MATRICOLA) = N(STUDENTI), ovvero ogni studente ha dato almeno un esame

Join PK-FK con predicati locali: esempio 2

```
SELECT COUNT(*) -- Q1J            RESULT = 1285
FROM   ORDER_HEADER OH, ORDER_DETAILS OD
WHERE  OH.ORDER_NUMBER = OD.ORDER_NUMBER
AND    OD.PRODUCT_NUMBER = 90110;

SELECT COUNT(*) -- Q1             RESULT = 1285
FROM   ORDER_DETAILS OD
WHERE  OD.PRODUCT_NUMBER = 90110;
```

```
N(ORDER_HEADER)=53267
N(ORDER_DETAILS)=446023
```

```
SELECT COUNT(*) - Q2J             RESULT = 5763
FROM   ORDER_HEADER OH, ORDER_DETAILS OD
WHERE  OH.ORDER_NUMBER = OD.ORDER_NUMBER
AND    OH.SALES_BRANCH_CODE = 20;

SELECT COUNT(*) - Q2              RESULT = 1062
FROM   ORDER_HEADER OH
WHERE  OH.SALES_BRANCH_CODE = 20;
```

- La stima di DB2 per il numero di tuple che Q2J conta sarebbe (1062/53267)*446023 = 8892

Ottimizzazione di query su singola relazione

- Se la query contiene un'unica relazione nella clausola FROM dobbiamo valutare solo proiezioni e selezioni (+ eventuali raggruppamenti ed operazioni aggregate)

- Ci sono 4 possibili soluzioni
  - Scansione sequenziale
  - Uso di un solo indice (eventualmente clustered)
  - Uso di più indici
  - Uso solo di un indice (index-only plan = non si accede ai dati)

Esempio

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

- Esistono un indice hash su vid, un B+-tree su anno e uno su (vid, anno, rivista)

Scansione sequenziale

- Si effettuano selezione e proiezione sul momento
- Costo = P(R)

Uso di un solo indice

- L'indice più selettivo è quello su vid
- Si leggono i record tramite i RID forniti dall'indice
- Viene valutato il predicato su anno ed effettuata la proiezione su rivista

Uso di più indici

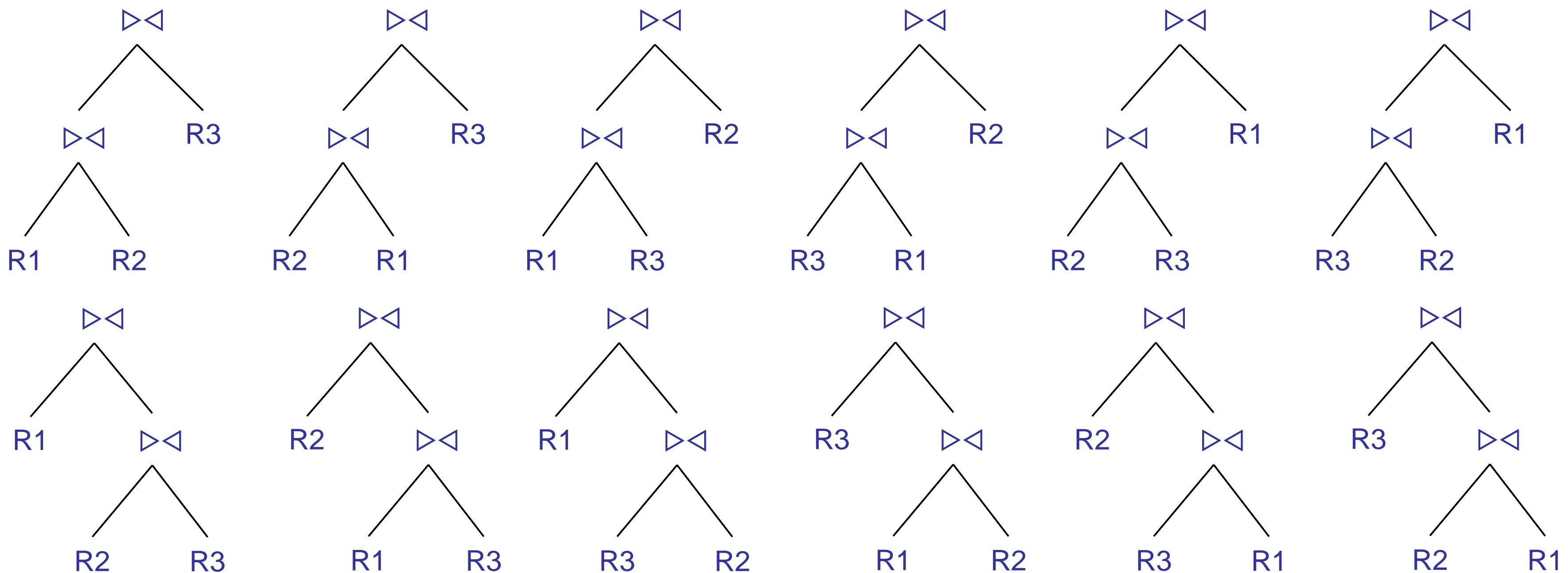
- Si usano sia l'indice su vid che quello su anno
- I RID ottenuti vengono intersecati
- Si leggono i record e si effettua la proiezione

Uso solo di un indice

- Si usa l'indice composto

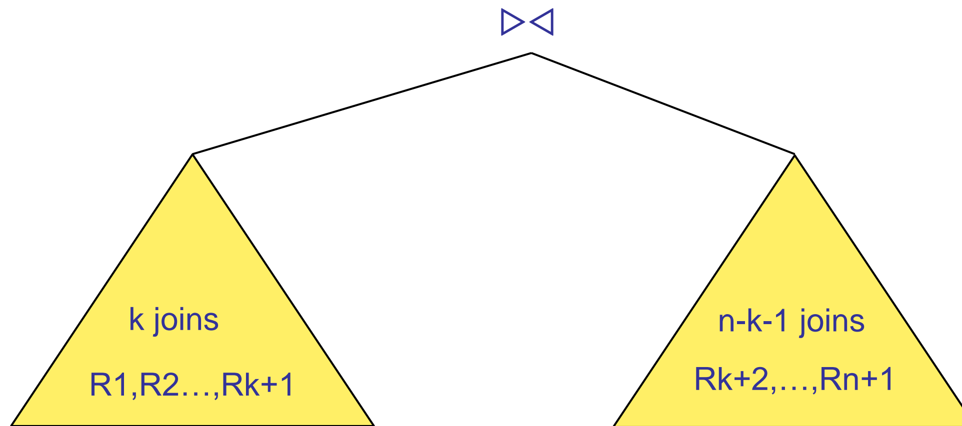
Query su più relazioni

- Il problema fondamentale nell'ottimizzazione di query su n+1 relazioni riguarda la scelta dell'ordine migliore in cui eseguire gli n join (binari), noto come join ordering problem
- Il problema nasce dall'enorme dimensione dello spazio di ricerca (esponenziale in n), dovuta al fatto che il join è commutativo e associativo
- Esempio: ci sono 12 possibili ordini diversi (o join trees) per eseguire 2 join:



Numero di join trees (1)

- Il numero di possibili alberi "di forma diversa" con n join, C_n, si ricava osservando che l'ultimo join può avere come operando sinistro un sottoalbero in cui sono presenti da 0 a n-1 join
  - Quindi $C_0 = 1$, $C_1 = 1$ e $C_2 = 2$



- Vale quindi la relazione:

$$C_n = \sum_{k=0}^{n-1} C_k \times C_{n-k-1}$$

Numero di join trees (2)

- La relazione vista è soddisfatta dai cosiddetti numeri Catalani, che in forma chiusa si esprimono come:

$$C_n = \frac{2n!}{(n+1)!\,n!}$$

NB: I Catalan numbers non traggono il loro nome dalla Catalogna, bensì devono il loro nome al matematico belga Eugène Charles Catalan che li studiò in relazione alla Torre di Hanoi. Prima di lui furono usati da Euler

- Molte applicazioni in ambito combinatorio, ad es:
  - Nella moltiplicazione $x_1*x_2*x_3*\ldots*x_{n+1}$ in quanto modi si possono mettere le parentesi?
  - Un poligono convesso di n+2 lati si può suddividere in triangoli in C_n modi diversi
- Considerando poi le (n+1)! possibili permutazioni delle n+1 relazioni, i possibili join trees su n+1 relazioni sono quindi:

$$\mathbf{JT_{n+1}} = \frac{\mathbf{2n!}}{\mathbf{n!}}$$

Un po' di numeri...

- Se poi si considera che ogni join può essere implementato in m modi diversi, i numeri vanno moltiplicati per m^n per ottenere il numero di possibili piani di accesso

N. relazioni (n+1)	C_n	N. join trees $JT_{n+1} = C_n * (n+1)!$
2	1	2
3	2	12
4	5	120
5	14	1680
6	42	30,240
7	132	665,280
8	429	17,297,280
10	4862	17,643,225,600

Programmazione dinamica

- La tecnica più comunemente adottata per ridurre il numero di piani di accesso da enumerare si basa sul principio di ottimalità, che è alla base degli algoritmi di programmazione dinamica
  - E’ la tecnica usata, ad esempio, per trovare il percorso di costo minimo in un grafo da un nodo sorgente S a un nodo target T

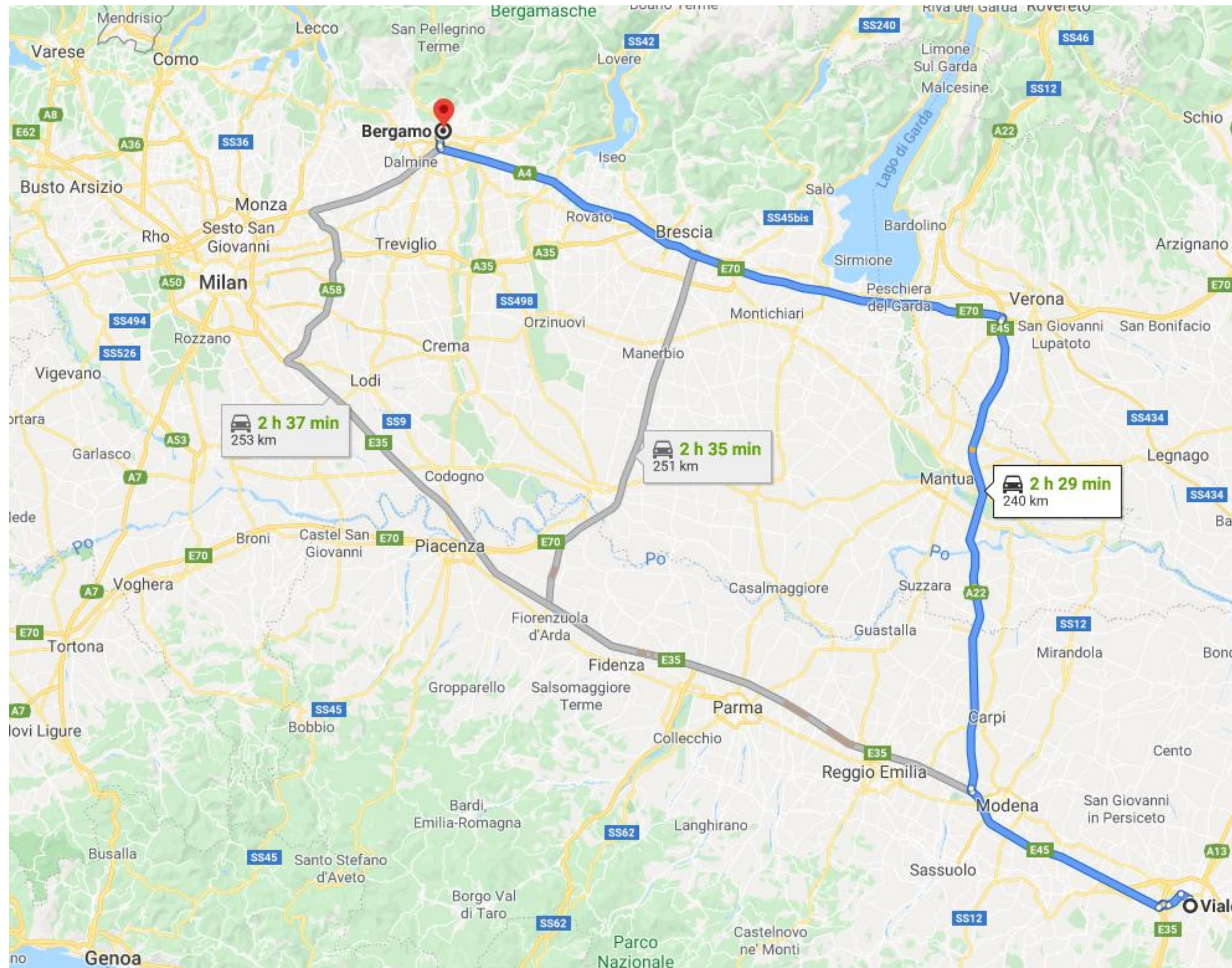
- Con riferimento alla terminologia usata nei grafi, il principio di ottimalità asserisce che

  dati 2 percorsi parziali P1 e P2 che hanno origine in S e arrivano entrambi in un nodo V, se costo(P1) < costo(P2), allora P2 non può essere esteso in modo tale da generare un percorso di costo minimo da S a T

- Quindi percorsi parziali come P2 possono essere ignorati…

Percorsi parziali ottimali…

- Supponete di dover andare da Ingegneria a Varese minimizzando i km
- Se passate da Bergamo, i 2 percorsi in grigio da BO a BG li potete scartare



Proprietà di piani di accesso

- Una semplice (ma importante!) osservazione è che ogni nodo di un piano di accesso può anche essere visto come la radice di un piano di accesso ("parziale", o "completo" se è il nodo radice)

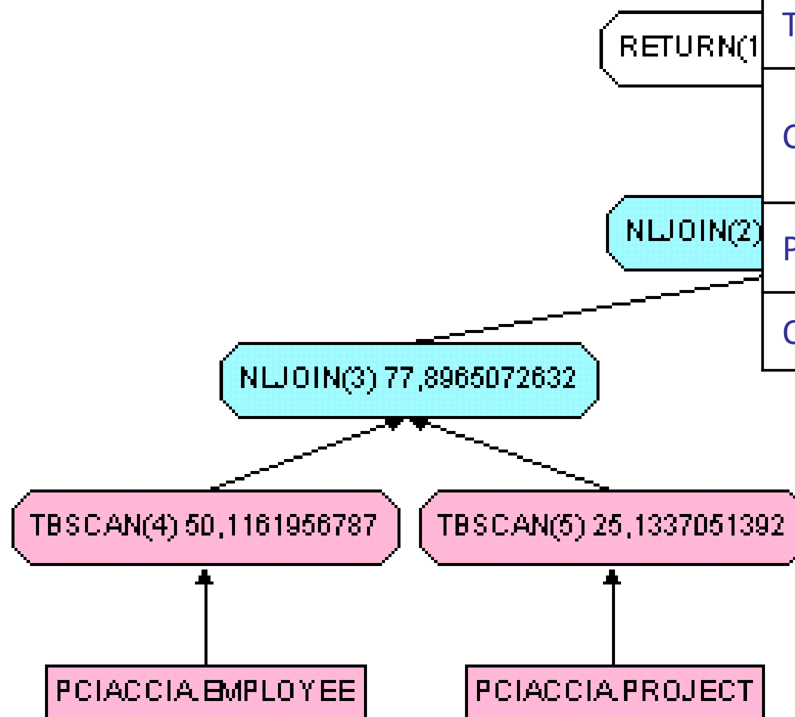
- Ogni piano di accesso (ovvero ogni nodo) è caratterizzabile con una serie di proprietà logiche e fisiche:
  - Relazioni elaborate
  - Schema (set di attributi)
  - Predicati applicati
  - Costo
  - Cardinalità
  - Ordine
  - ….

Proprietà di un nodo = f(proprietà dei figli, operatore del nodo)

Proprietà di un nodo: esempio

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

IDNodo	4
Cost	50.116…
Cardinality	42
Tables	{Employee}
Columns	{WorkDept, EmpNo}
Predicates	-
Order	EmpNo



IDNodo	3
Cost	77.896…
Cardinality	20
Tables	{Employee, Project}
Columns	{EmpNo, WorkDept, ProjNo}
Predicates	{E.EmpNo = P.RespEmp}
Order	EmpNo

IDNodo	5
Cost	25.133…
Cardinality	20
Tables	{Project}
Columns	{RespEmp, ProjNo}
Predicates	-
Order	-

Ottimizzazione con programmazione dinamica

- L'idea alla base dell'algoritmo di programmazione dinamica (DP) è costruire il piano di accesso ottimale (= a costo minimo) per n relazioni componendo i piani di accesso (parziali) ottimali costruiti per 1,2,....,n-1 relazioni

- Si procede quindi per passi:

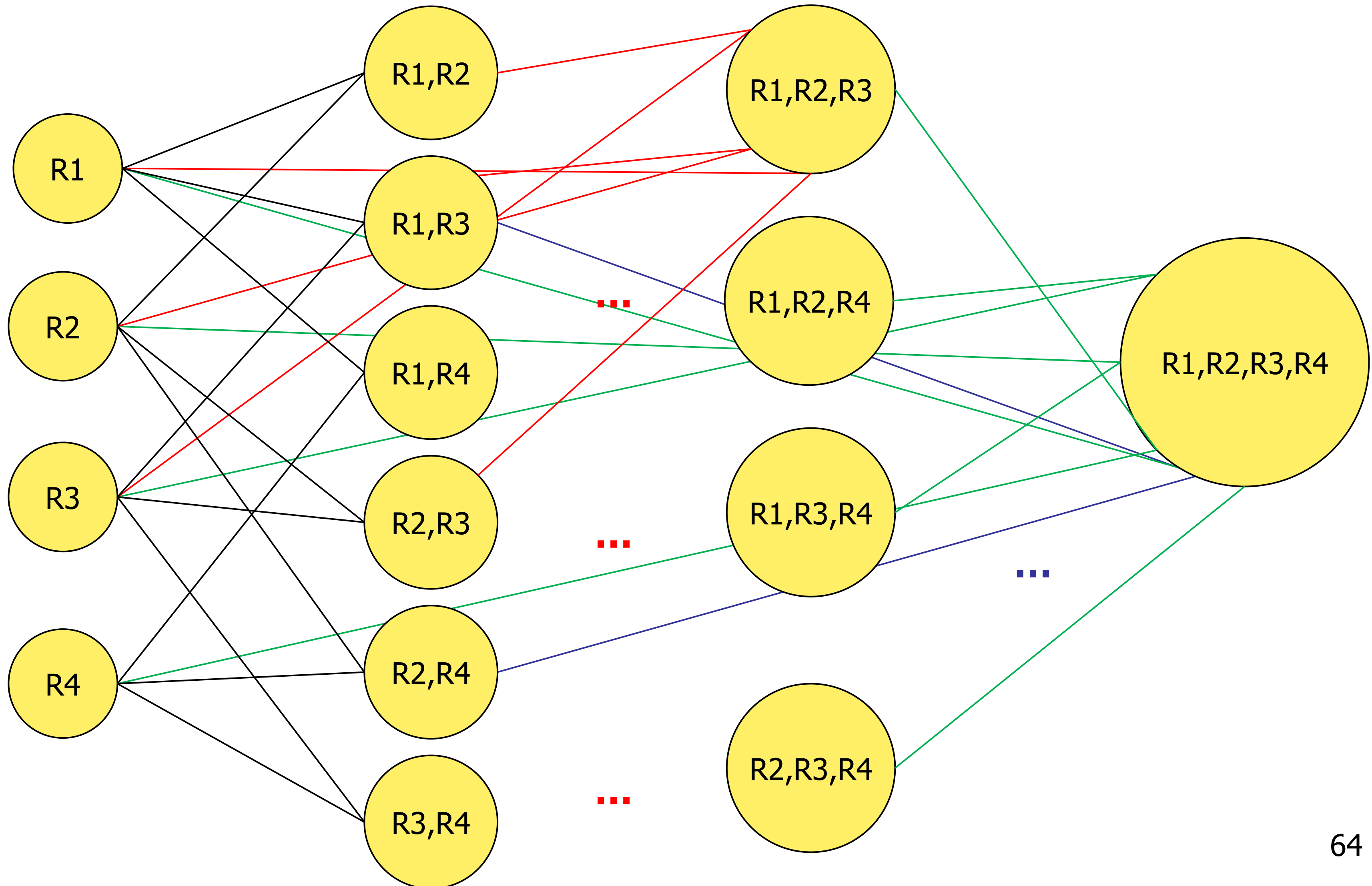
Algoritmo DP

Passo 1: determina per ogni relazione Ri il piano parziale migliore

Passo k (k = 2,...,n): per ogni sottoinsieme di k relazioni determina il piano migliore, a partire dai soli piani selezionati nei passi precedenti, ossia combinando i piani ottimali per i e k-i relazioni (i=1,...,k-1)

Programmazione dinamica: n = 4

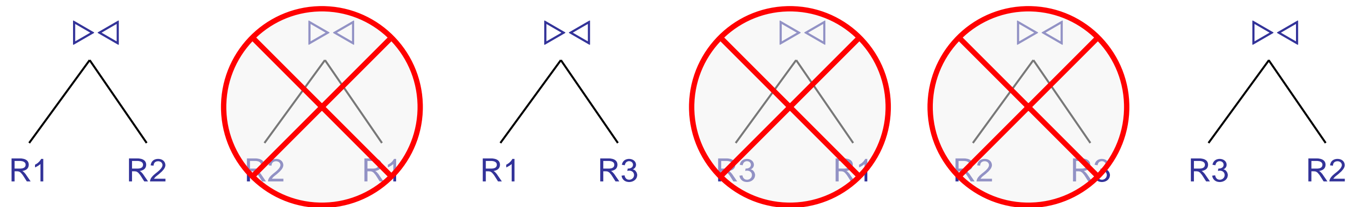
- Schema di condivisione dei piani di accesso parziali



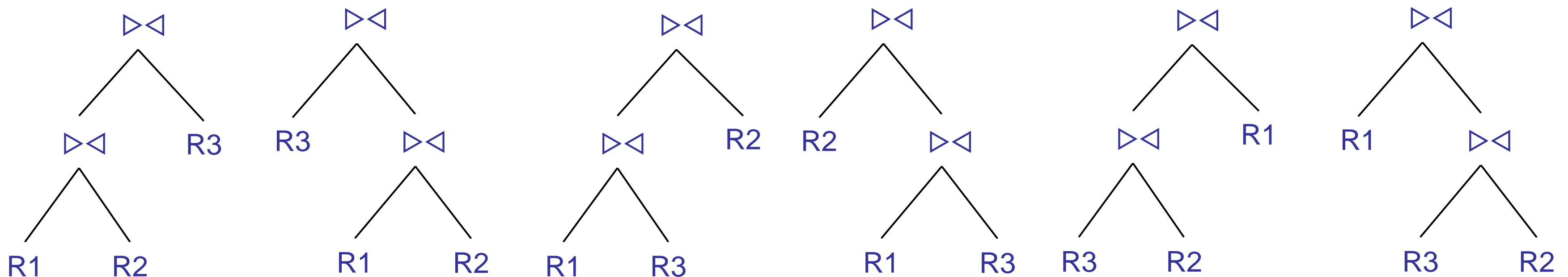
Programmazione dinamica: n = 3

I join trees considerati dall'algoritmo sono:

- Passo 1: R1 R2 R3
- Passo 2:



  - Per ogni coppia di relazioni si mantiene uno dei 2 join trees, quello che per ipotesi costa meno
- Passo 3: si compongono i join trees mantenuti nei passi precedenti e tra questi si determina quello a costo minimo



Di quanto si riducono i join trees

- Il numero di join trees enumerati dall'algoritmo DP, JT_DP_n, è ancora esponenziale in n, ma molto minore di JT_n
  - Si ricorda che i join trees non sono propriamente access plans, in quanto mancano sempre i diversi metodi di join
- Dato un sottoinsieme S di k relazioni (k = 2,…,n), per determinare il piano ottimale per S si devono combinare con un join 2 join trees su i e k-i relazioni (i=1,…,k-i)
  - Per il caso k = 1 si hanno complessivamente n join trees
- L'operando sinistro del join introdotto da S è quindi qualsiasi sottoinsieme proprio di S, e di questi ce ne sono 2^k-2
- Poiché di sottoinsiemi S di k relazioni ce ne sono $\binom{n}{k}$ si ha:

$$\mathbf{JT_DP_n} = n + \sum_{k=2}^{n} \binom{n}{k} \left(2^k - 2\right) = \mathbf{3^n} - \mathbf{2^{n+1}} + (\mathbf{n} + \mathbf{1})$$

Ancora un po' di numeri…

N. relazioni (n)	N. join trees JT_n	join trees enumerati JT_DP_n
2	2	4
3	12	15
4	120	54
5	1680	185
6	30,240	608
7	665,280	1,939
8	17,297,280	6,058
10	17,643,225,600	57,012

Programmazione dinamica: un esempio (1)

- Vediamo un esempio completo con n = 4, in cui per semplicità il costo viene calcolato come la somma delle cardinalità intermedie
  - Ad es. per il join tree $((R1 \rhd\lhd R2) \rhd\lhd R3) \rhd\lhd R4$, se $N(R1 \rhd\lhd R2) = 2000$, e $N((R1 \rhd\lhd R2) \rhd\lhd R3) = 3500$ il costo è 5500
  - Quindi commutare gli operandi di un join è ininfluente

- La query da ottimizzare è:

```
SELECT *
FROM   R1,R2,R3,R4
WHERE  R1.K1 = R2.K1 AND R3.K3 = R4.K3
AND    R2.A = R3.A;
```

TABLE	N
R1	2000
R2	10000
R3	500
R4	20000

COLUMN	NK
R2.A	1000
R3.A	250

- I primi 2 join sono PK-FK con Ki chiave di Ri
- Il join tra R2 e R3 è molti-a-molti, e produce N(R2)*N(R3)/max{NK(R2.A),NK(R3.A)} = 5000 tuple

Programmazione dinamica: un esempio (2)

- Al primo passo si riportano semplicemente le cardinalità delle singole relazioni. Per tutti i piani il costo è nullo:

ID	Tables	Card	Cost	Join tree
1	{R1}	2,000	**0**	R1
2	{R2}	10,000	**0**	R2
3	{R3}	500	**0**	R3
4	{R4}	20,000	**0**	R4

```
TABLE        N
------------
R1        2000
R2       10000
R3         500
R4       20000

COLUMN      NK
------------
R2.A      1000
R3.A       250
```

Programmazione dinamica: un esempio (3)

- Al passo 2 si generano i 6 subset di 2 relazioni
- Data la funzione di costo i join tree da confrontare sono equivalenti:

ID	Tables	Card	Cost	Join tree
…	…	…	…	…
5	{R1,R2}	10,000	**10,000**	R1▷◁R2
6	{R1,R3}	1,000,000	**1,000,000**	R1▷◁R3
7	{R1,R4}	40,000,000	**40,000,000**	R1▷◁R4
8	{R2,R3}	5,000	**5,000**	R2▷◁R3
9	{R2,R4}	200,000,000	**200,000,000**	R2▷◁R4
10	{R3,R4}	20,000	**20,000**	R3▷◁R4

```
TABLE        N
------------
R1        2000
R2       10000
R3         500
R4       20000

COLUMN      NK
------------
R2.A      1000
R3.A       250
```

Programmazione dinamica: un esempio (4)

- Al passo 3 si generano i 4 subset di 3 relazioni, ognuno dei quali si può ottenere in 6 modi diversi
  - Ad es. per {R1,R2,R3} i 6 modi (2 a 2 equivalenti) sono:
    - (R1▷◁R2)▷◁R3, R3▷◁(R1▷◁R2)
    - (R1▷◁R3)▷◁R2, R2▷◁(R1▷◁R3)
    - (R2▷◁R3)▷◁R1, R1▷◁(R2▷◁R3)

```
TABLE          N
------------
R1          2000
R2         10000
R3           500
R4         20000

COLUMN        NK
-----------
R2.A        1000
R3.A         250
```

ID	Tables	Card	Cost	Join tree
…	…	…	…	…
5	{R1,R2}	10,000	**10,000**	R1▷◁R2
6	{R1,R3}	1,000,000	**1,000,000**	R1▷◁R3
7	{R1,R4}	40,000,000	**40,000,000**	R1▷◁R4
8	{R2,R3}	5,000	**5,000**	R2▷◁R3
9	{R2,R4}	200,000,000	**200,000,000**	R2▷◁R4
10	{R3,R4}	20,000	**20,000**	R3▷◁R4

ID	Tables	Card	Cost	Join tree
11	{R1,R2,R3}	5,000	**10,000**	(R2▷◁R3)▷◁R1
12	{R1,R2,R4}	200,000,000	**200,010,000**	(R1▷◁R2)▷◁R4
13	{R1,R3,R4}	40,000,000	**40,020,000**	(R3▷◁R4)▷◁R1
14*	{R2,R3,R4}	200,000	**205,000**	(R2▷◁R3)▷◁R4

* La cardinalità del piano 14 è stimata usando la formula vista, ovvero:
N(R4)*N(R2▷◁R3)/max{NK(R4.K3,NK(R2▷◁R3.K3)} =
20,000*5,000/max{≤500,500} = 200,000

Programmazione dinamica: un esempio (5)

- All'ultimo passo si genera la soluzione finale, confrontando $14 = 2^4-2$ alternative (2 a 2 equivalenti)
  - Basta sommare i costi dei piani di accesso componenti
  - Il risultato finale contiene 200,000 tuple, ma questo non influenza la scelta (non è un risultato intermedio, e sarebbe comunque comune a tutti i piani di accesso)

outer	Cost outer	inner	Cost inner	Total cost	Join tree
(R2▷◁R3)▷◁R1	**10,000**	R4	**0**	**10,000**	((R2▷◁R3)▷◁R1) ▷◁R4
…	…	…	…	…	…
(R2▷◁R3)	**5,000**	(R1▷◁R4)	**40,000,000**	**40,005,000**	(R2▷◁R3)▷◁(R1▷◁R4)

- E' facile verificare che tutti gli altri piani hanno un costo maggiore
  - Quelli che combinano 3 relazioni con l'ultima costano già di più
  - Tra quelli che combinano 2 relazioni con le altre due, solo R2▷◁R3 (ID 8) potrebbe fare meglio, ma si combina con ID 7 che costa 40 milioni (ultima riga nella tabella)

Riduzione dello spazio di ricerca

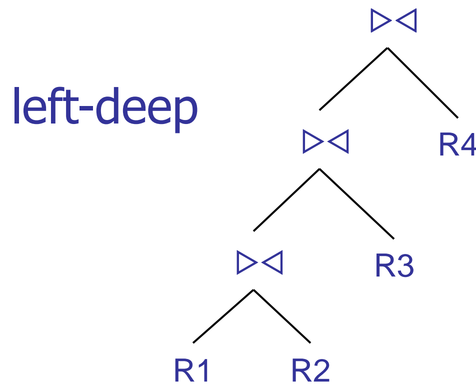
- Poiché il numero di possibili piani di accesso è comunque esponenziale in n, e lo spazio richiesto dall'algoritmo DP è anch'esso esponenziale in n, per query con molti join può essere opportuno limitare lo spazio di ricerca, facendo uso di tecniche euristiche che permettano comunque di trovare una buona soluzione in tempi accettabili
- Le più comuni euristiche adottate sono:
  - Non considerare piani che includono prodotti Cartesiani
  - Considerare solo left-deep join trees
- Il tutto è tipicamente controllabile via SQL

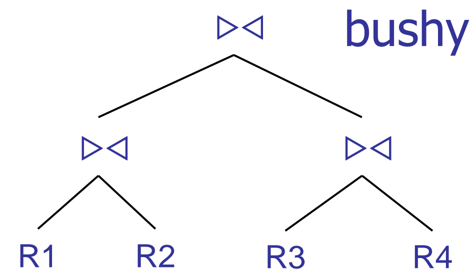
IBM DB2

```
SET CURRENT QUERY OPTIMIZATION = n
```

Left-deep join trees

- In un left-deep join tree il figlio destro (inner) di un nodo di join è sempre una relazione del DB. Gli altri alberi si chiamano bushy





- In questo modo è molto più semplice generare piani che lavorano in pipeline (si può usare index nested loops)
- Euristica comunemente adottata a partire da System R

Esempio

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

- I join tree left-deep e senza prodotti Cartesiani sono solo 4 (= 12 - 6 bushy - 2 con prodotti Cartesiani):
  - (R ▷◁ S) ▷◁ V
  - (S ▷◁ R) ▷◁ V
  - (R ▷◁ V) ▷◁ S
  - (V ▷◁ R) ▷◁ S

Progr. dinamica con left-deep join trees

- Il numero di left-deep join trees è $JTLD_n = n!$, quindi comunque grande
- L'algoritmo di programmazione dinamica si applica praticamente invariato, ma per generare un piano ottimale per un insieme S di k relazioni si considerano solo join trees tali che l'operando sinistro è un sottoinsieme di S con k-1 relazioni
- Quindi per S consideriamo solo k casi (anziché 2^k-2)!
- Il numero di left-deep join trees da enumerare è quindi:

$$\mathbf{JTLD_DP_n} = \sum_{k=1}^{n} \binom{n}{k} k = \mathbf{n} \times \mathbf{2^{n-1}}$$

n	JT_DP_n	$JTLD_n$	$JTLD_DP_n$
2	4	2	4
3	15	6	12
4	54	24	32
5	185	120	80
6	608	720	192
7	1,939	5,040	448
8	6,058	40,320	1,024
10	57,012	3,628,800	5,120

Greedy join enumeration

- Per diminuire ulteriormente lo spazio di ricerca è possibile adottare una strategia di ricerca "greedy" che, ad ogni passo, seleziona il join ancora da eseguire che ha costo minimo
  - A parità di costo, quello che produce il minor numero di record
- Es: n=4

Passo 1: si seleziona il join a costo migliore, ad es.: R2 ▷◁ R3

Passo 2: si sceglie il join a costo minore tra:

(R2 ▷◁ R3) ▷◁ R1, R1 ▷◁ (R2 ▷◁ R3)

(R2 ▷◁ R3) ▷◁ R4, R4 ▷◁ (R2 ▷◁ R3)

R1 ▷◁ R4, R4 ▷◁ R1

Passo 3: Se si è scelto, ad es.: R1 ▷◁ (R2 ▷◁ R3), l'ultimo join è

[R1 ▷◁ (R2 ▷◁ R3)] ▷◁ R4 o R4 ▷◁ [R1 ▷◁ (R2 ▷◁ R3)]

Ordini significativi

- L'ordine delle tuple di un nodo è detto significativo se può influenzare le operazioni ancora da compiere (o il risultato finale)

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

- Supponendo di eseguire prima il join tra V e R, il solo possibile ordine significativo è su sid, perché può influenzare il join con S
- Se si esegue prima il join tra S e R i possibili ordini significativi sono invece:
  - vid (può influenzare il join con V)
  - snome (semplifica l'ORDER BY)

- Si noti che se si avessero le tuple ordinate su snome, cantina (che risolve l'ORDER BY) quest'ordine sarebbe più significativo di quello sul solo snome

Ordini significativi: pruning di piani parziali

- La presenza di uno o più ordini significativi richiede di estendere l'algoritmo DP, in quanto il solo costo non è più sufficiente a garantire l'ottimalità
- La seguente osservazione è alla base della determinazione efficiente del piano ottimale quando si considerano anche ordini significativi:

  Un piano di accesso (parziale) AP per un sottoinsieme di relazioni S, il cui risultato è ordinato secondo un ordine significativo 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

- Si dice che AP' "domina" AP
  - Le proprietà di AP' dominano quelle di AP

- L'algoritmo DP viene esteso mantenendo per ogni sottoinsieme di relazioni S:
  - Il piano non ordinato di costo minimo
  - Un piano di costo minimo per ogni ordine significativo

Pruning con ordini significativi: esempio

- La query include la clausola `ORDER BY R.B`

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

AP	546
Cost	216
Cardinality	25
Tables	{R,S}
Columns	{R.A,R.B,S.C}
Predicates	{R.J=S.J, R.A > 5}
Order	-

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

- AP1127 domina AP224, che viene eliminato
- AP546 non domina AP1127, perché non è ordinato (mentre AP 1127 lo è)

- Quindi si mantengono AP546 e AP1127

SET CURRENT QUERY OPTIMIZATION

- E' una direttiva che permette di controllare il lavoro svolto dall'ottimizzatore

0 Basic rewrite rules, greedy join enumeration, only nested loops join
1 Frequent values statistics are not used, greedy join enumeration, subset of rewrite rules
2 All statistics are used, almost all rewrite rules are applied, greedy join enumeration
3 DP enumeration (left-deep & no Cartesian products), index ANDing
5 (default) More complex rewrite rules
7 Similar to 5 but without the heuristic rules
9 A maximal amount of optimization is performed to generate an access plan.

- In realtà DB2 è più flessibile per quanto riguarda i left-deep trees: è possibile specificare il n. massimo di relazioni che formano l'operando destro di un join
  - Left-deep: massimo = 1
  - Ma può anche essere 2, 3, …

Ordine di enumerazione di piani di accesso

- L'algoritmo di programmazione dinamica opera in modo breadth-first (per "livelli" = num. relazioni considerate)
- In realtà questo non è strettamente necessario. Qualsiasi ordine di enumerazione va bene, purché sia rispettato il seguente vincolo:

Un piano di accesso (parziale) per un sottoinsieme S di relazioni può essere generato solo se sono stati determinati i piani di accesso ottimali per tutti i sottoinsiemi propri di S

- Nel caso di left-deep trees è possibile anche una generazione alternativa, di tipo best-first, in cui viene espanso il piano di accesso parziale più "promettente" (tipicamente: costo minore)
- Si estende di conseguenza il concetto di dominazione:
  - Perché AP' domini AP va anche verificato che le relazioni elaborate da AP' includano quelle elaborate da AP

    eguaglianza $\Rightarrow$ inclusione

Esempio di generazione best-first (1)

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

- Sono disponibili indici B+-tree unclustered su R.vid, R.sid, S.livello e S.sid
- Gli unici algoritmi di join disponibili sono il nested loop e l'index nested loop
  - Quindi non ci sono ordini significativi

- 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.livello)=20, L(S.livello)=3, NK(S.sid)=200, L(S.sid)=5

Esempio di generazione best-first (2)

- Passo 1: si valuta il metodo di accesso di costo minimo per ogni relazione
  - Relazione R:
    - Cardinalità risultato = 2000/100 = 20

      (AP1) Costo sequenziale = 400

      (AP2) Costo indice R.vid = 20/100 + Φ(2000/100,400)=1+20=21
  - Relazione S:
    - Cardinalità risultato = 200/20 = 10

      (AP3) Costo sequenziale = 100

      (AP4) Costo indice S.livello = 3/20 + Φ(200/20,100)=1+10=11

ID	Tables	Card	Cost	Preds	AccessPlan
AP4	{S}	10	**11**	S.livello=4	IXSCAN(S)
AP2	{R}	20	**21**	R.vid=100	IXSCAN(R)

Esempio di generazione best-first (3)

ID	Tables	Card	Cost	Preds	AccessPlan
AP4	{S}	10	**11**	S.livello=4	IXSCAN(S)
AP2	{R}	20	**21**	R.vid=100	IXSCAN(R)

- Si espande AP4 considerando il join con R
  - Nested loop:
    - Costo di partenza = 11
    - Costo per ogni loop = 21 (indice R.vid)

    (AP5) Costo = 11 + 10 x 21 = 221
  - Index nested loop (indice R.sid):
    - Costo di partenza = 11
    - Costo per ogni loop = 25/200 + Φ(2000/200,400)=1+10 = 11

    (AP6) Costo = 11 + 10 x 11 = 121

ID	Tables	Card	Cost	Preds	AccessPlan
AP2	{R}	20	**21**	R.vid=100	IXSCAN(R)
AP6	{R,S}	20	**121**	S.livello=4, R.vid=100,S.sid=R.sid	IXSCAN(S) NLJOIN IXSCAN(R)

Esempio di generazione best-first (4)

ID	Tables	Card	Cost	Preds	AccessPlan
AP2	{R}	20	**21**	R.vid=100	IXSCAN(R)
AP6	{R,S}	20	**121**	S.livello=4, R.vid=100,S.sid=R.sid	IXSCAN(S) NLJOIN IXSCAN(R)

- Si espande AP2
  - Nested loop:
    - Costo di partenza = 21
    - Costo per ogni loop = 11 (indice S.livello)

    (AP7) Costo = 21 + 20 x 11 = 241
  - Index nested loop (indice S.sid):
    - Costo di partenza = 21
    - Costo per ogni loop = 1 + 1 = 2 (sid chiave di S)

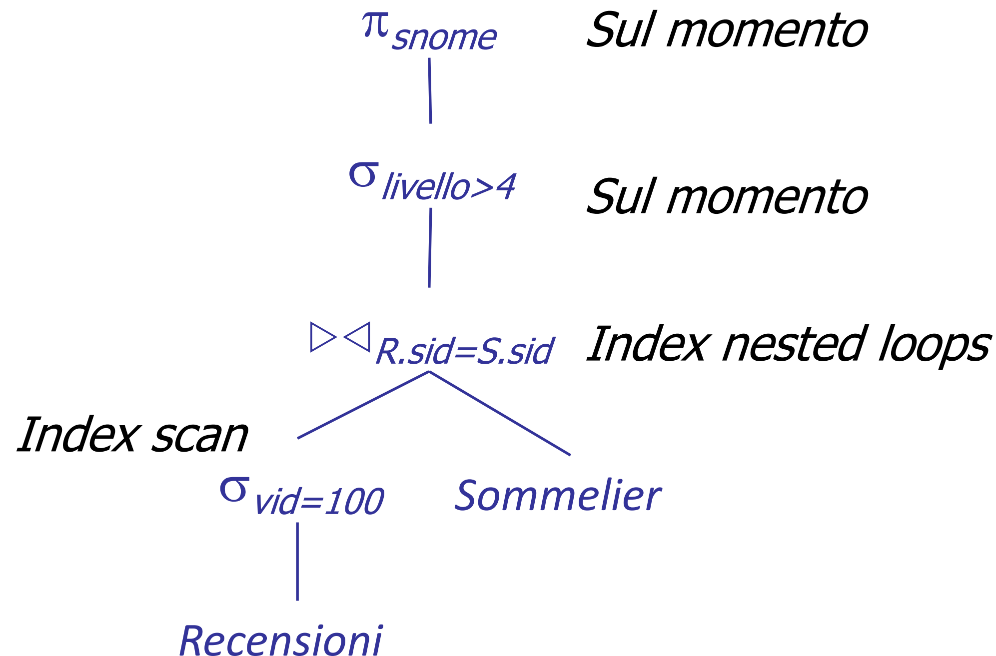
    (AP8) Costo = 21 + 20 x 2 = 61

ID	Tables	Card	Cost	Preds	AccessPlan
AP8	{R,S}	20	**61**	S.livello=4, R.vid=100,S.sid=R.sid	IXSCAN(R) NLJOIN IXSCAN(S)
AP6	{R,S}	20	**121**	S.livello=4, R.vid=100,S.sid=R.sid	IXSCAN(S) NLJOIN IXSCAN(R)

Esempio di generazione best-first (5)

- Poiché il piano di accesso in testa alla coda, AP8, è completo ci si ferma

ID	Tables	Card	Cost	Preds	AccessPlan
AP8	{R,S}	20	**61**	S.livello=4, R.vid=100,S.sid=R.sid	IXSCAN(R) NLJOIN IXSCAN(S)
AP6	{R,S}	20	**121**	S.livello=4, R.vid=100,S.sid=R.sid	IXSCAN(S) NLJOIN IXSCAN(R)



Esempio con ordini significativi (1)

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

- Esistono indici B^+-tree unclustered su R.sid e R.vid
- Esistono indici B^+-tree clustered su S.sid e V.vid
  - Ordini significativi possibili: sid, vid

- Si ignorano i costi di accesso agli indici
  - N(R)=5000, P(R)=1000, NK(R.sid)=200, NK(R.vid)=100
  - N(S)=200, P(S)=100, NK(S.livello)=10, NK(S.sid)=200
  - N(V)=100, P(V)=20, NK(V.vid)=100, NK(V.vnome)=20

Esempio con ordini significativi (2)

- Si trovano i metodi di accesso di costo minimo alle relazioni, conservando anche quelli che generano ordinamenti utili

  (AP1) Scansione sequenziale 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 sequenziale di S:
  - Costo = 100, risultati = 200/10 = 20, ordine: sid

  (AP5) Scansione sequenziale di V :
  - Costo = 20, risultati = 100/20 = 5, ordine: vid

ID	Tables	Card	Cost	Order	Preds	AccessPlan
AP5	{V}	5	**20**	vid	V.vnome='Merlot'	TBSCAN(V)
AP4	{S}	20	**100**	sid	S.livello=4	TBSCAN(S)
AP1	{R}	5000	**1000**	-	-	TBSCAN(R)
AP2	{R}	5000	**5000**	sid	-	IXSCAN(R)
AP3	{R}	5000	**5000**	vid	-	IXSCAN(R)

Esempio con ordini significativi (3)

ID	Tables	Card	Cost	Order	Preds	AccessPlan
AP5	{V}	5	**20**	vid	V.vnome='Merlot'	TBSCAN(V)
AP4	{S}	20	**100**	sid	S.livello=4	TBSCAN(S)
AP1	{R}	5000	**1000**	-	-	TBSCAN(R)
AP2	{R}	5000	**5000**	sid	-	IXSCAN(R)
AP3	{R}	5000	**5000**	vid	-	IXSCAN(R)

- Si espande AP5 considerando solo il join V ▷◁ R (V ▷◁ S è prodotto cartesiano)
  - Index nested loop:
    - Costo per ogni loop = 5000/100

      (**AP6**) Costo = 20 + 5 x 50 = 270
    - Ordine: vid
  - Merge-scan:
    - Costo di merge = 5000 (indice unclustered su R.vid)

      (AP7) Costo = 20 + 5000 = 5020
    - Ordine: vid
  - Cardinalità risultato = 5000/20 = 250
- AP1 e AP3 sono dominati da AP6

ID	Tables	Card	Cost	Order	Preds	AccessPlan
AP4	{S}	20	**100**	sid	S.livello=4	TBSCAN(S)
AP6	{V,R}	250	**270**	vid	V.vnome='Merlot',R.vid=V.vid	TBSCAN(V) NLJOIN IXSCAN(R)
AP2	{R}	5000	**5000**	sid	-	IXSCAN(R)

Esempio con ordini significativi (4)

- Si espande AP4 con il join S ▷◁ R

ID	Tables	Card	Cost	Order	Preds	AccessPlan
AP4	{S}	20	**100**	sid	S.livello=4	TBSCAN(S)
AP6	{V,R}	250	**270**	vid	V.vnome='Merlot', R.vid=V.vid	TBSCAN(V) NLJOIN IXSCAN(R)
AP2	{R}	5000	**5000**	sid	-	IXSCAN(R)

- Index nested loop:
  - Costo per ogni loop = 5000/200 = 25

  (**AP8**) Costo = 100 + 20 x 25 = 600
  - Ordine: sid
- Merge-scan:
  - Costo di merge = 5000 (indice unclustered su R.sid)

  (AP9) Costo = 100 + 5000 = 5100
  - Ordine: sid
- Dimensione risultato = 5000/10 = 500

- AP8 domina AP2

ID	Tables	Card	Cost	Order	Preds	AccessPlan
AP6	{V,R}	250	**270**	vid	V.vnome='Merlot',R.vid=V.vid	TBSCAN(V) NLJOIN IXSCAN(R)
AP8	{S,R}	500	**600**	sid	S.livello=4, R.sid=S.sid	TBSCAN(S) NLJOIN IXSCAN(R)

Esempio con ordini significativi (5)

ID	Tables	Card	Cost	Order	Preds	AccessPlan
AP6	{V,R}	250	**270**	vid	V.vnome='Merlot', R.vid=V.vid	TBSCAN(V) NLJOIN IXSCAN(R)
AP8	{S,R}	500	**600**	sid	S.livello=4, R.sid=S.sid	TBSCAN(S) NLJOIN IXSCAN(R)

- Si espande AP6 con il join con S : (V ▷◁ R) ▷◁ S
  - Index nested loop:
    - Costo per ogni loop = 1 (sid chiave di S)

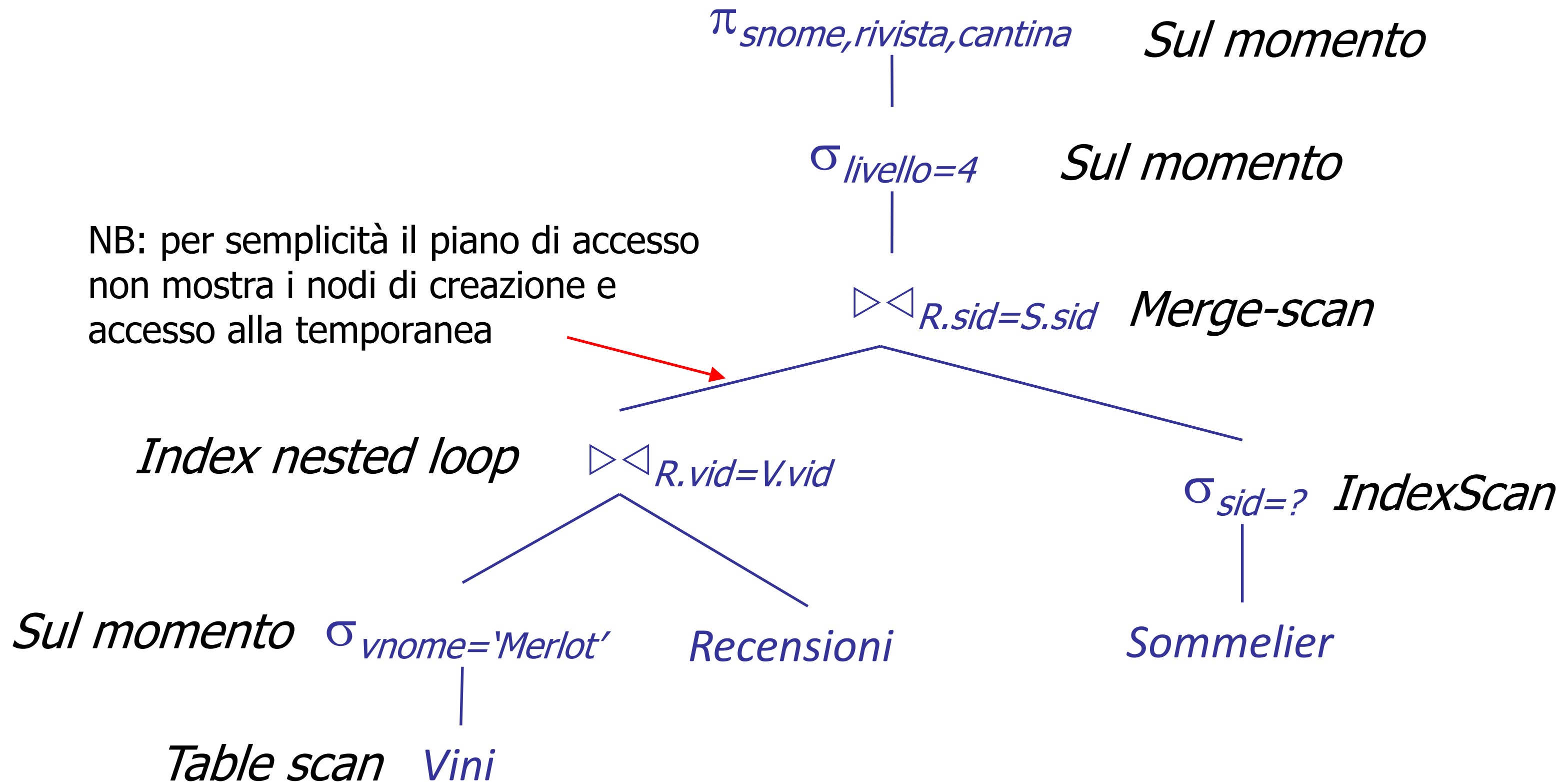
    (AP10) Costo = 270 + 250 x 1 = 520
  - Merge-scan (con sort dell'esterna su sid):
    - Costo di sort = 4 * P(temp) (sorting su sid della rel. temporanea)
    - Costo di merge = P(temp) + P(temp) + 100

    (**AP11**) Costo = 270 + 4 * 10 + 10 + 10 + 100 = 430
  - Dimensione risultato = 250/10 = 25
- AP11 domina AP8

ID	Tables	Card	Cost	Order	Preds	AccessPlan
AP11	{V,R,S}	25	**430**	sid	V.vnome='Merlot',R.vid=V.vid, S.livello=4,R.sid=S.sid	SORT(TBSCAN(V) NLJOIN IXSCAN(R)) MSJOIN TBSCAN(S)

Esempio con ordini significativi (6)

ID	Tables	Card	Cost	Order	Preds	AccessPlan
AP11	{V,R,S}	25	**430**	sid	V.vnome='Merlot',R.vid=V.vid, S.livello=4,R.sid=S.sid	SORT(TBSCAN(V) NLJOIN IXSCAN(R)) MSJOIN TBSCAN(S)



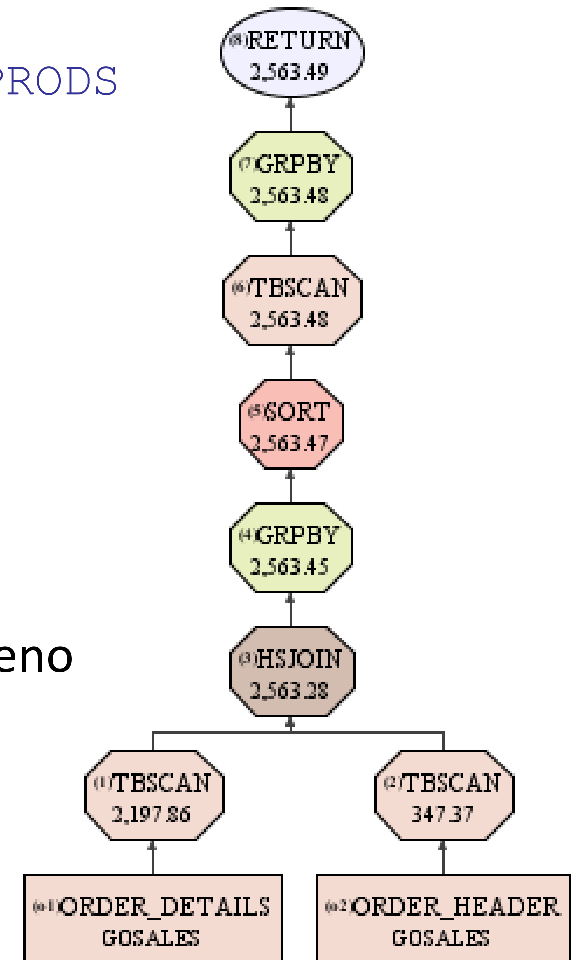
E i raggruppamenti?

- Se la query include una clausola `GROUP BY`, questa viene valutata dopo tutti i join previsti
  - Unico approccio contemplato dai DBMS del XX secolo…
- In realtà è possibile in alcuni casi fare il push-down del `GROUP BY`, ed eseguirlo prima di uno o più join
  - Condizione da verificare: i valori delle funzioni aggregate non devono cambiare
  - Quindi: o si scarta tutto il gruppo, oppure nessuna tupla del gruppo viene scartata
  - Questo è il caso in cui si aggrega e poi si fanno solo join FK-PK, in cui la FK fa parte degli attributi di raggruppamento
- O anche eseguire dei ragguppamenti parziali…
- In entrambi i casi l'obiettivo è ridurre la cardinalità degli input dei join

Push-down del GROUP BY (1)

- Si consideri la seguente query, in cui il join (del tipo FK-PK) serve solo a scartare alcuni ordini (e quindi alcuni gruppi):

```
SELECT   OD.ORDER_NUMBER, COUNT(*) AS NUM_PRODS
FROM     ORDER_DETAILS OD, ORDER_HEADER OH
WHERE    OD.ORDER_NUMBER = OH.ORDER_NUMBER
AND      OH.RETAILER_NAME = 'AcquaVerde'
GROUP BY OD.ORDER_NUMBER
```

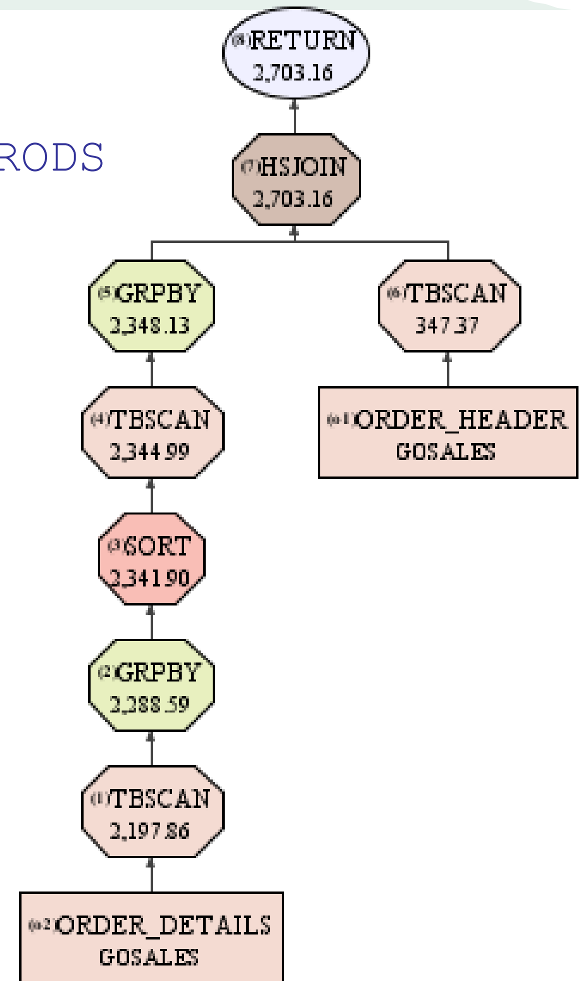


- NUM_PRODS può essere calcolato correttamente prima del join per tutti gli ordini, alcuni dei quali vengono eliminati dopo il join
- … ma DB2 non contempla questa possibilità, nemmeno al massimo livello di ottimizzazione (9)
  - Si noti il grpby eseguito in 2 fasi, la prima delle quali è basata su hashing
- Forse perché questo è comunque il piano migliore?

Push-down del GROUP BY (2)

- La seguente query è equivalente alla prima:

```
SELECT  OD.ORDER_NUMBER, COUNT(*) AS NUM_PRODS
FROM    ORDER_DETAILS OD
GROUP BY OD.ORDER_NUMBER
HAVING OD.ORDER_NUMBER IN (
   SELECT  OH.ORDER_NUMBER
   FROM    ORDER_HEADER OH
   WHERE   OH.RETAILER_NAME = 'AcquaVerde')
```



- Il costo del nuovo piano è stimato di poco superiore al precedente, ma il fatto che quest'ultimo non sia scelto anche per questa nuova formulazione mostra che DB2 non riconosce come equivalenti le due query
- Quindi: verificare sempre cosa un DBMS sa effettivamente fare, e sulla base di ciò prestare attenzione a come le query vengono formulate

Conclusioni

- L'ottimizzazione di query SQL è un problema complesso, per il quale sono state proposte innumerevoli soluzioni
- I moderni DBMS hanno a disposizione molte strategie e strumenti che permettono di affrontare il problema in modo da massimizzare le prestazioni del sistema
  - Noi abbiamo visto il "core" del problema
- Nonostante ciò la soluzione "finale" non esiste ancora, e sempre nuovi aspetti vengono presi in considerazione e analizzati
  - Il recente lavoro [LGM+15] ben riassume i problemi che i moderni ottimizzatori devono affrontare
- Ad esempio: la "parametric query optimization" affronta il problema di come compilare una query in cui i parametri (ad es. selettività dei predicati) sono noti solo a run-time, caso tipico per le query che vengono eseguite molte volte all'interno di programmi applicativi
  - Possibile soluzione: generare tanti piani di accesso, e scegliere a run-time quello per il quale i valori dei parametri sono più vicini a quelli effettivi

L'insieme dei possibili piani ottimali

- PICASSO è un tool di visualizzazione che permette di vedere le regioni di ottimalità dei piani di accesso, in funzione delle selettività dei predicati. Ogni colore si riferisce a un diverso piano

  N. Reddy, J.R. Haritsa: Analyzing Plan Diagrams of Database Query Optimizers. VLDB 2005