Elaborazione di interrogazioni

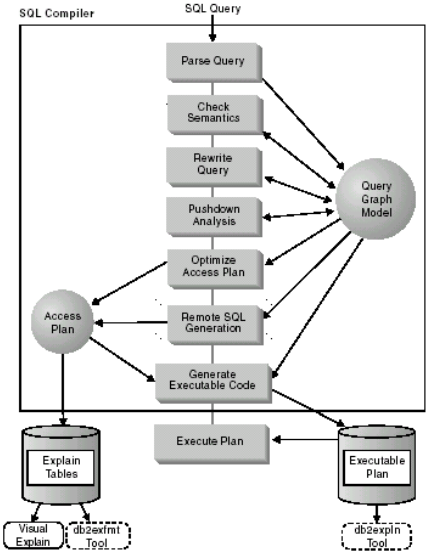
Sistemi Informativi L-B

Home Page del corso:
http://www-db.deis.unibo.it/courses/SIL-B/

Versione elettronica: interrogazioni.pdf

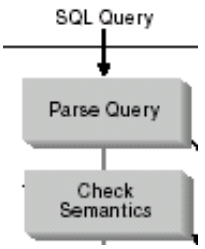
Passi del processo di elaborazione

- L'elaborazione di una query SQL comporta l'esecuzione di una serie di passi, riassunti in figura con riferimento a DB2 UDB
- Ogni sistema adotta soluzioni specifiche in fase di elaborazione delle query, noi vediamo gli aspetti che hanno validità generale



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 tutti gli oggetti interessati (relazioni, attributi, ecc.)
- La fase di check semantico 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
- A tale scopo il processore SQL fa riferimento ai cataloghi, in cui sono memorizzate tutte le informazioni necessarie



Check semantici e uso dei cataloghi (1)

- 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 (2)

- … e supponiamo di avere la seguente query:

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

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

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

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


Autorità e privilegi

- Nei DBMS SQL ogni operazione deve essere autorizzata, ovvero l'utente che esegue l'operazione deve avere i privilegi necessari.
- I privilegi vengono concessi e revocati per mezzo delle istruzioni **GRANT** e **REVOKE**
- Un principio fondamentale è che un utente che ha ricevuto un certo privilegio può a sua volta accordarlo ad altri utenti solo se è stato esplicitamente autorizzato a farlo
- Mediante **GRANT** e **REVOKE** si controllano anche le autorità, ovvero il diritto ad eseguire azioni amministrative di un certo tipo
  - Ad esempio, se si ha l'autorità **SYSADM** (che include anche quella di **DBADM**) è possibile passare ad altri utenti l'autorità **DBADM** (Database Administrator Authority):

    ```
    GRANT DBADM ON DATABASE TO Pippo WITH GRANT OPTION;
    ```

    in cui la clausola **WITH GRANT OPTION** autorizza l'utente Pippo a passare l'autorità ad altri utenti

GRANT (1)

- Il formato dell'istruzione GRANT per assegnare privilegi su table e view è:

```
GRANT { ALL | < lista di privilegi > }
ON [ TABLE ] <table name>
TO { <lista di utenti e gruppi> | PUBLIC }
[ WITH GRANT OPTION ]
```

- I privilegi possibili includono quello "master" di **CONTROL** e quelli di **ALTER**, **DELETE**, **INSERT**, **SELECT**, **INDEX**, **REFERENCES** e **UPDATE** (per questi ultimi due si può anche specificare una lista di attributi)
  - **ALL** conferisce tutti i privilegi, ma non **CONTROL**
  - **PUBLIC** concede i privilegi specificati a tutti gli utenti, inclusi quelli futuri

GRANT (2)

- CONTROL: comprende tutti i privilegi (su una view sono solo SELECT, INSERT, DELETE e UPDATE). Inoltre permette di conferire tali privilegi ad altri utenti; può essere conferito solo da qualcuno che ha autorità SYSADM o DBADM
- ALTER: attribuisce il diritto di modificare la definizione di una tabella
- DELETE: attribuisce il diritto di cancellare righe di una tabella
- INDEX: attribuisce il diritto di creare un indice sulla tabella
- INSERT: attribuisce il diritto di inserire righe nella tabella
- REFERENCES: attribuisce il diritto di creare chiavi secondarie che referenziano la tabella
- SELECT: attribuisce il diritto di eseguire query sulla tabella/vista e di definire VIEW
- UPDATE: attribuisce il diritto di modificare righe della tabella/vista

- Per eseguire una query, è necessario avere il privilegio di SELECT o di CONTROL su tutte le table e le view referenziate dalla query

GRANT: esempi

- Paperino autorizza Pippo e Topolino a leggere la relazione Employee e a modificare i valori di Salary; inoltre concede loro di passare questo privilegio ad altri utenti:

```
Paperino>  GRANT SELECT, UPDATE(Salary)
           ON TABLE Employee TO USER Pippo, USER Topolino
           WITH GRANT OPTION
```

- … e Pippo ne approfitta subito:

```
Pippo>     GRANT SELECT
           ON TABLE Employee TO Pluto
```

- Pluto può eseguire query su Employee, ma non aggiornamenti; inoltre non può passare lo stesso privilegio ad altri

REVOKE

- Il formato dell'istruzione REVOKE per revocare privilegi su table è:

```
REVOKE { ALL | < lista di privilegi > }
ON [ TABLE ] <table name>
FROM { <lista di utenti e gruppi> | PUBLIC }
```

  - A differenza del GRANT, per eseguire REVOKE bisogna avere l'autorità **SYSADM** o **DBADM**, oppure il privilegio di **CONTROL** sulla relazione
  - Il REVOKE non agisce a livello di singoli attributi; pertanto non si possono revocare privilegi di **UPDATE** solo su un attributo e non su altri (per far ciò è quindi necessario revocarli tutti e poi riassegnare solo quelli che si vogliono mantenere)

REVOKE: esempi

- Se Pippo, che non ha autorità DBADM o SYSADM, né CONTROL su Employee, prova ad eseguire:

```
Pippo>     REVOKE SELECT
           ON TABLE Employee FROM Pluto
```

  si verifica un errore
- Viceversa, se Paperino ha autorità `DBADM` ed esegue

```
Paperino>  REVOKE SELECT
           ON TABLE Employee FROM Pippo, Topolino
```

  né Pippo né Topolino possono più eseguire query su Employee, ma continuano a poter aggiornare Salary
- Se Pippo avesse creato una view su Employee, questa diventerebbe "inoperativa", ovvero non più utilizzabile
- Si noti che Pluto mantiene il privilegio SELECT su Employee
- SQL-92 ha una gestione del REVOKE più complessa, che include anche effetti di revoca dei privilegi "in cascata" (in cui quindi Pluto perderebbe il privilegio di SELECT)

Cataloghi: TABAUTH

- Il catalogo dei privilegi a livello di table è TABAUTH, che contiene una tupla per ogni coppia (GRANTOR, GRANTEE)

SQL Catalog	SQL attribute	Descrizione
TABAUTH	GRANTOR	Chi ha concesso i privilegi
TABAUTH	GRANTEE	Chi li ha ricevuti
TABAUTH	GRANTEETYPE	U = User; G = Group
TABAUTH	TABSCHEMA	Nome dello schema
TABAUTH	TABNAME	Nome della table o vista
TABAUTH	CONTROLAUTH	Privilegio di CONTROL: Y = Sì; N = No (implica tutti gli altri)
TABAUTH	SELECTAUTH	Privilegio di SELECT: Y = Sì; G = con GRANT OPTION; N = No
TABAUTH	UPDATEAUTH	Privilegio di UPDATE: Y = Sì; G = con GRANT OPTION; N = No
TABAUTH	…	…

Cataloghi: COLAUTH

- Il catalogo per i privilegi sui singoli attributi è COLAUTH

SQL Catalog	SQL attribute	Descrizione
COLAUTH	GRANTOR	Chi ha concesso i privilegi
COLAUTH	GRANTEE	Chi li ha ricevuti
COLAUTH	GRANTEETYPE	U = User; G = Group
COLAUTH	TABSCHEMA	Nome dello schema
COLAUTH	TABNAME	Nome della table o vista
COLAUTH	COLNAME	Nome dell'attributo
COLAUTH	PRIVTYPE	U = UPDATE; R = REFERENCES
COLAUTH	GRANTABLE	G = con GRANT OPTION; N = No

Autorizzazioni: verifica

- Supponiamo di avere la seguente query, eseguita da Pippo:

```
Pippo>      SELECT    EmpNo
            FROM      MySchema.Employee
```

- Il DBMS prima esegue una query del tipo:

```
SELECT    DEFINER
FROM      SYSCAT.TABLES
WHERE     TABSCHEMA = 'MySchema' AND TABNAME = 'Employee';
```

per verificare se Pippo è il "definer" di Employee; in caso contrario, si verifica se Pippo ha il privilegio di SELECT su Employee:

```
SELECT    *
FROM      SYSCAT.TABAUTH
WHERE     GRANTEE = 'Pippo'
  AND     TABSCHEMA = 'MySchema' AND TABNAME = 'Employee'
  AND     SELECTAUTH IN ('Y','G');
```

- Controlli simili vengono eseguiti per le altre istruzioni, incluse GRANT e REVOKE

Riscrittura di interrogazioni

- 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
- 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, e quindi non è possibile fornire soluzioni di validità generale (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%'
```

Unnesting (1)

- Il passaggio a una forma senza subquery alle volte è di immediata comprensione; ad esempio la seguente query:

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

viene riscritta come:

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

- Si noti che non è necessaria l'opzione `DISTINCT` in quanto `WorkDept` è foreign key

Unnesting (2)

- Altre volte la trasformazione è meno facile da interpretare. Ad esempio, la query:

```
SELECT  E.EmpNo
FROM    Employee E
WHERE   NOT EXISTS ( SELECT  *
                     FROM   Emp_Act EA
                     WHERE  E.EmpNo = EA.EmpNo )
```

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 (1)

- Il DBMS "ragiona" sfruttando la presenza di vincoli per semplificare le query.
- Ad esempio, se `EmpNo` è una chiave:

```
SELECT  DISTINCT EmpNo
FROM    Employee
```

si riscrive più semplicemente come:

```
SELECT  EmpNo
FROM    Employee
```

che ha il vantaggio di non comportare una (inutile!) operazione di rimozione delle tuple duplicate dal risultato

Uso dei vincoli (2)

- Anche il vincolo di Foreign Key può essere efficientemente sfruttato:

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

viene riscritta come:

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

- Se `EA.EmpNo` ammettesse valori nulli, bisognerebbe aggiungere

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

per rispettare la semantica della query

- In questo esempio l'utente stesso potrebbe scrivere la query in forma semplificata; nel prossimo esempio ciò non è possibile, ma può farlo solo il DBMS

Uso dei vincoli (3)

- 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 su tale vista:

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

  in cui chi esegue la query non ha privilegio `SELECT` né su `Employee` né su `Department`, ma solo su `People`

- Il DBMS può però semplificare la query:

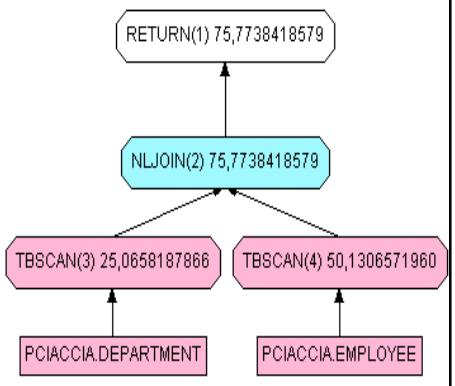
```
SELECT  LastName,FirstName
FROM    Employee
WHERE   WorkDept IS NOT NULL
```

  in cui la clausola `WHERE` serve solo se `WorkDept` ammette valori nulli

Piani di accesso

- Al termine della fase di riscrittura viene eseguita l'ottimizzazione vera e propria, anche detta "ottimizzazione basata sui costi"
- In questa fase, facendo anche uso delle informazioni statistiche sui dati, viene scelto il modo "più economico" per eseguire la query, ovvero il modo che complessivamente presenta un costo minimo tra tutte le alternative possibili
- Ogni "modo" di eseguire una query è detto piano di accesso, e si compone di una serie di operatori connessi ad albero
- Le foglie del piano di accesso sono le relazioni di base presenti nella query riscritta
- Gli altri nodi sono operatori che agiscono su 1 o 2 insiemi di tuple in input e producono 1 insieme di tuple in output

```
SELECT *
FROM   Department, Employee
WHERE  WorkDept = DeptNo
```



Operatori relazionali

- Parlando di operatori è opportuno distinguere tra:
  - Operatori logici (es. Join, Sort)
    - Sono un'estensione di quelli dell'algebra relazionale; svolgono una determinata funzione e producono un insieme di tuple con certe proprietà (ad es. ordinate sul campo DeptNo)
  - Operatori fisici (es. join nested-loops, o NLJOIN in DB2)
    - Sono implementazioni specifiche di un operatore logico; in funzione dei valori delle statistiche, dei parametri di sistema (es. dimensione del buffer pool) e della logica dell'algoritmo è possibile associare ad ogni operatore fisico un costo di esecuzione
- Vediamo prima come si implementano due operatori fondamentali, ovvero il Sort e il Join, quindi trattiamo gli aspetti legati all'esecuzione di un piano di accesso composto da un numero arbitrario di operatori

L'operatore Sort

- L'operatore Sort, nella sua versione di base, ha come

  INPUT: un insieme di tuple

  OUTPUT: lo stesso insieme di tuple ordinato su un attributo A (o su una combinazione di attributi A1,A2,…,An)
- Esistono ovviamente diverse varianti, ad esempio:
  - Se richiesto, si possono eliminare i duplicati durante l'esecuzione del Sort
  - Se alcuni attributi in input non servono nell'ouput, si possono eliminare durante l'esecuzione del Sort

```
SELECT    DISTINCT LastName
FROM      Employee
ORDER BY  LastName
```

Sort: esempio

attributi nel risultato — attributi di ordinamento

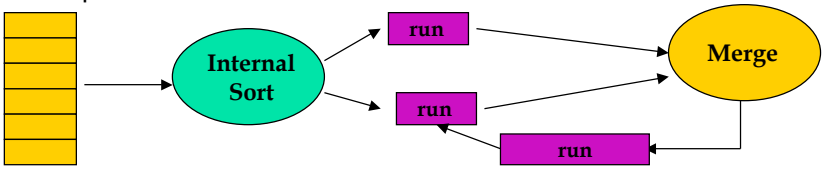
Matricola	CodiceFiscale	Cognome	Nome	DataNascita
210629323	BNCGRG78L21A944K	Bianchi	Giorgio	21/07/1978
216635467	RSSNNA78A53A944V	Rossi	Anna	13/01/1978
160239654	VRDMRC79H21F839X	Verdoni	Marco	21/06/1979
214842132	VRDCRL79H20G125J	Verdi	Carlo	20/06/1979
2006431216	RSSDRA78M10A944V	Rossi	Dario	10/08/1978

```
SELECT Cognome, Nome,
  Matricola, DataNascita
FROM Studenti
ORDER BY Cognome, Nome
```

Matricola	Cognome	Nome	DataNascita
210629323	Bianchi	Giorgio	21/07/1978
216635467	Rossi	Anna	13/01/1978
2006431216	Rossi	Dario	10/08/1978
214842132	Verdi	Carlo	20/06/1979
160239654	Verdoni	Marco	21/06/1979

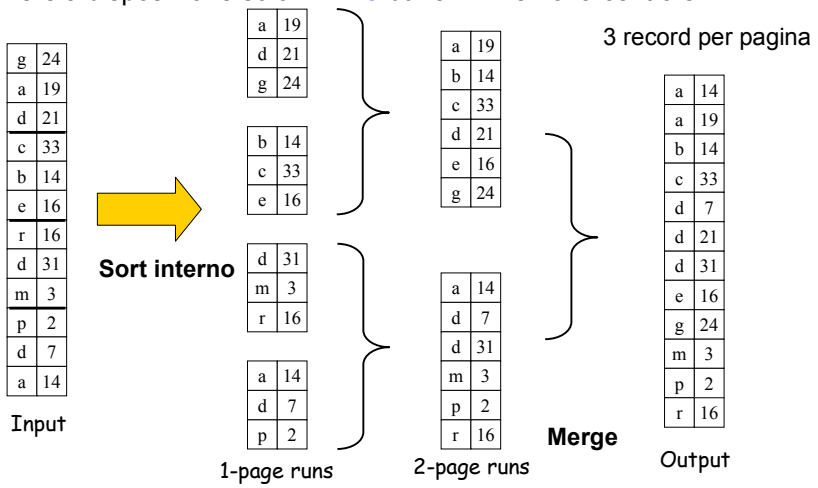
Sort: implementazione

- L'algoritmo più comunemente utilizzato dai DBMS è quello detto di **Merge Sort a Z vie** (Z-way Sort-Merge o anche External Merge Sort)
- Supponiamo di dover ordinare un input che consiste di un file di NP pagine e di avere a disposizione solo NB < NP buffer in memoria centrale
- L'algoritmo opera in **2 fasi**:
  - **Sort interno**: si leggono una alla volta le pagine del file; i record di ogni pagina vengono ordinati facendo uso di un algoritmo di sort interno (es. Quicksort); ogni pagina così ordinata, detta anche "**run**", viene scritta su disco in un file temporaneo
  - **Merge**: operando uno o più passi di fusione, le run vengono fuse, fino a produrre un'unica run



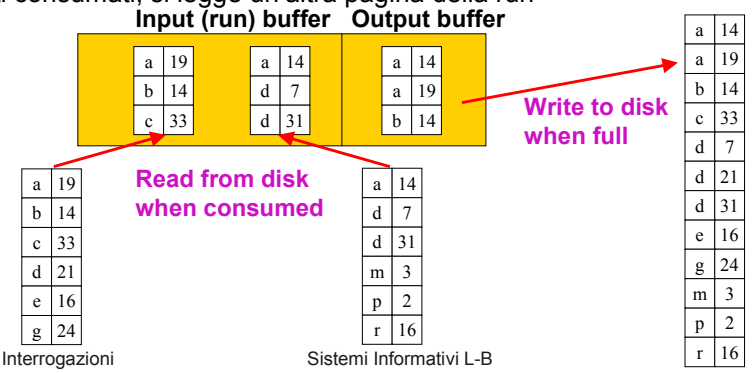
Z-way Merge Sort: caso base

- Per semplicità consideriamo il caso base a Z = 2 vie, e supponiamo di avere a disposizione solo NB = 3 buffer in memoria centrale



Fusione delle run

- Nel caso base Z = 2 si fondono 2 run alla volta
- Con NB = 3, si associa un buffer a ognuna delle run, il terzo buffer serve per produrre l'output, 1 pagina alla volta
- Si legge la prima pagina da ciascuna run e si può quindi determinare la prima pagina dell'output; quando tutti i record di una pagina di run sono stati consumati, si legge un'altra pagina della run



Caso base: complessità

- Consideriamo per semplicità solo il numero di operazioni di I/O
- Nel caso base Z = 2 e NB = 3 si può osservare che:
  - Nella fase di sort interno si leggono e si riscrivono NP pagine
  - Ad ogni passo di merge si leggono e si riscrivono NP pagine
    - Il numero di passi fusione è pari a $\lceil \log_2 NP \rceil$, in quanto ad ogni passo il numero di run si dimezza
- Il costo complessivo è pertanto pari a **$2 * NP * (\lceil \log_2 NP \rceil + 1)$**

Esempio: per ordinare NP = 8000 pagine sono necessarie 224000 operazioni di I/O; se ogni I/O richiede 20 msec, il sort richiede 4480 secondi, ovvero circa 1h 15 minuti!

- In realtà se NP non è una potenza di 2 il numero effettivo di I/O è leggermente minore di quello calcolato, in quanto in uno o più passi di fusione può capitare che una run non venga fusa con un'altra
- Nell'ultimo passo di fusione si può evitare di scrivere su disco, ma si possono produrre direttamente le tuple in output man mano che vengono generate (*piped sort*)

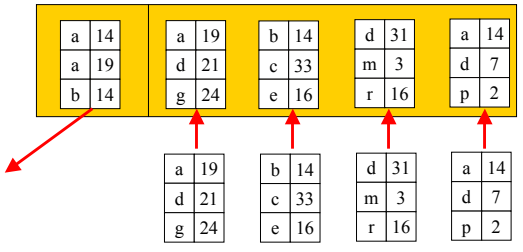
Z-way Sort-Merge: caso generale

- Una prima osservazione è che nel passo di sort interno, avendo a disposizione NB buffer, si possono ordinare NB pagine alla volta (anziché una sola), il che porta a un costo di **$2 * NP * (\lceil \log_2 \lceil NP/NB \rceil \rceil + 1)$**

Esempio: con NP = 8000 pagine e NB = 3 si hanno ora 208000 I/O

- Miglioramenti sostanziali si possono ottenere se, avendo NB > 3 buffer a disposizione, si fondono NB – 1 run alla volta (1 buffer è per l'output)
- In questo caso il numero di passi di fusione è logaritmico in NB – 1, ovvero è pari a **$2 * NP * (\lceil \log_{NB-1} \lceil NP/NB \rceil \rceil + 1)$**

Esempio:
con NP = 8000 pagine
e NB = 11 si hanno 64000 I/O,
per un tempo stimato
pari a 1280 sec



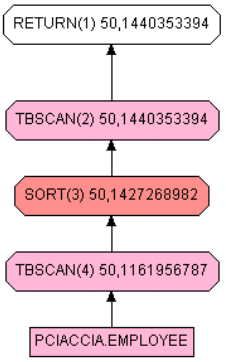
Esempio di piano di accesso con Sort

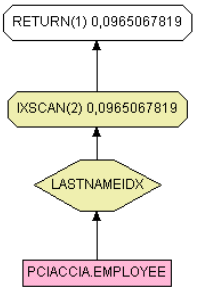
- Data la query

```
SELECT    LastName
FROM      Employee
ORDER BY  LastName
```

il piano di accesso a destra rappresenta l'unica alternativa ragionevole se non esiste un indice su `LastName`

- Se si definisse un indice su `LastName`, l'ottimizzatore genererebbe invece il seguente piano di accesso

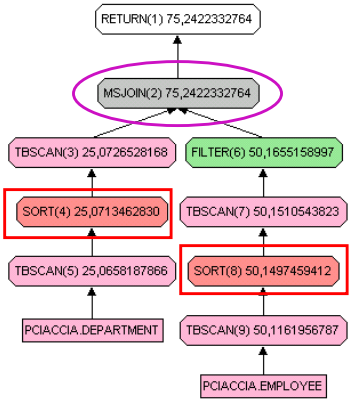




Utilità del Sort

- Oltre che per ordinare le tuple, il Sort può essere utilizzato per:
  - Query in cui compare l'opzione **DISTINCT**, ovvero per eliminare i duplicati
  - Query che contengono la clausola **GROUP BY**
- In entrambi i casi alcuni DBMS fanno uso di operatori che usano tecniche hash per "raggruppare" le tuple con valori uguali su tutti gli attributi (nel caso di DISTINCT) o solo su alcuni (nel caso di GROUP BY)
- Come vedremo, il Sort può anche essere considerato dall'ottimizzatore come passo preliminare per eseguire un Join

```
SELECT *
FROM   Department, Employee
WHERE  WorkDept = DeptNo
```



L'operatore Join

- L'operatore Join, nella sua versione di base, ha come
  INPUT: due insiemi di tuple
  OUTPUT: un insieme in cui ogni tupla è ottenuta combinando, sulla base di uno o più predicati di join, una tupla del primo insieme con una tupla del secondo insieme

- Anche per il Join vi sono diverse varianti:
  - Nel caso di Outer Join non è richiesta la presenza di "matching tuples"
  - Gli attributi non richiesti in output possono essere eliminati quando si produce il risultato del Join

```
SELECT      E.Empno, D.DeptNo
FROM        Employee E, Department D
WHERE       E.WorkDept = D.DeptNo
  AND       E.EmpNo <> D.MgrNo
```


Operatori fisici per il Join

- Esistono moltissime implementazioni del Join, che mirano a sfruttare al meglio le risorse del sistema e le (eventuali) proprietà degli insiemi di tuple in ingresso per evitare di eseguire tutti i possibili NR(R) * NR(S) confronti
- Le implementazioni più diffuse si riconducono ai seguenti operatori fisici:
  - Nested Loops Join (DB2: NLJOIN)
  - Merge Scan Join (DB2: MSJOIN)
  - Hash Join (DB2: HSJOIN)
- Si noti che, benché logicamente il Join sia commutativo, dal punto di vista fisico vi è una chiara distinzione, che influenza anche le prestazioni, tra operando sinistro (o "esterno", "outer") e operando destro (o "interno", "inner")
- Per semplicità nel seguito parliamo di "relazione esterna" e "relazione interna" per riferirci agli input del Join, ma va ricordato che in generale l'input può derivare dall'applicazione di altri operatori

Nested Loops Join

- Date 2 relazioni in input R e S tra cui sono definiti i predicati di join PJ, e supponendo che R sia la relazione esterna, l'algoritmo opera come segue

  Per ogni tupla t_R in R**:**

  { Per ogni tupla t_S in S**:**

  { se la coppia (t_R, t_S) soddisfa PJ

  allora aggiungi (t_R, t_S) al risultato **} }**

R

A	B
22	a
87	s
45	h
32	b

S

A	C	D
22	z	8
45	k	4
22	s	7
87	s	9
32	c	3
45	h	5
32	g	6

PJ: R.A = S.A

A	C	D	B
22	z	8	a
22	s	7	a
87	s	9	s
45	k	4	h
45	h	5	h
32	c	3	b
32	g	6	b

Nested Loops Join: costi

- Il costo di esecuzione dipende dallo spazio a disposizione in memoria centrale
- Nel caso base in cui vi sia 1 buffer per R e 1 buffer per S, bisogna leggere 1 volta R e NR(R) volte S, ovvero tante volte quante sono le tuple della relazione esterna, per un totale di **NP(R) + NR(R) * NP(S)** I/O
- Se è possibile allocare NP(S) buffer per la relazione interna il costo si riduce a **NP(R) + NP(S)**
- Nel caso in cui i buffer per S siano NB(S) < NP(S):
  - Se si usa LRU (Least Recently Used) ogni richiesta di una pagina di S dà luogo a un page fault, e quindi il costo è ancora **NP(R) + NR(R) * NP(S)**
  - Se si usa MRU (Most Recently Used), per la prima tupla di R si leggono tutte le NP(S) pagine di S, ma per le altre se ne devono leggere da disco solo NP(S) – NB(S), e quindi il costo si riduce considerevolmente

Nested Loops: esempio

- Se si fa il join tra Department e Employee (WorkDept = DeptNo) e si ha:
  NP(Department) = 20 NR(Department) = 100
  NP(Employee) = 1000 NR(Employee) = 10000
  e considerando il caso base (1 buffer per ciascuna relazione):

Department esterna:
NP(Department) + NR(Department) * NP(Employee) = **100020 I/O**

Employee esterna:
NP(Employee) + NR(Employee) * NP(Department) = **201000 I/O**

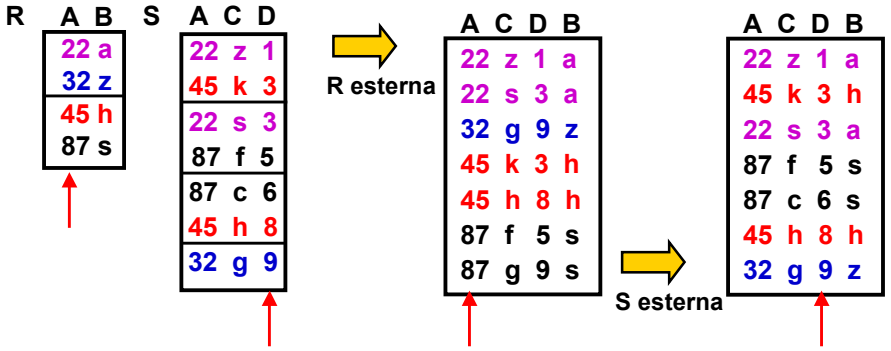
- Pertanto, con 20 msec per I/O, nel primo caso sarebbero comunque necessari circa 33 min!

Scelta della relazione esterna

- La scelta della relazione esterna può dipendere da vari fattori:
  - La possibilità di mantenere in memoria centrale tutta la relazione interna porta a scegliere come esterna la relazione con più pagine
  - Se i buffer sono in numero limitato, trascurando il costo di lettura della relazione esterna, si sceglierà R come esterna e S come interna se **NR(R) * NP(S) < NR(S) * NP(R)**, ovvero se:
    NR(R) / NP(R) < NR(S) / NP(S)
    che corrisponde a dire che le tuple di R sono più grandi di quelle di S
  - Vi sono però altre considerazioni da fare, non meno importanti…

Nested loops: cammini di accesso

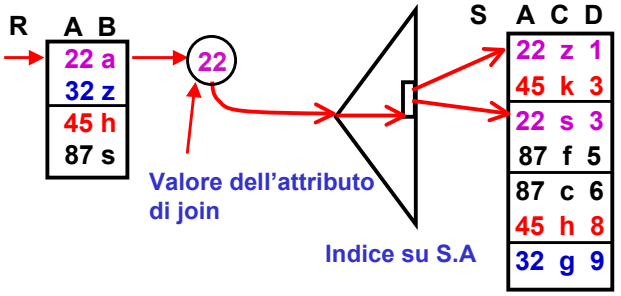
- L'ordine con cui vengono generate le tuple del risultato coincide con l'ordine eventualmente presente nella relazione esterna



Pertanto se l'ordine che si genera è "interessante", ad esempio perché la query contiene `ORDER BY R.A`, la scelta della relazione esterna può risultarne influenzata

Nested loops: presenza di indici

- Data una tupla della relazione esterna R, la scansione completa della relazione interna S può essere sostituita da una scansione basata su un indice costruito sugli attributi di join di S, secondo il seguente schema:



- L'accesso alla relazione interna mediante indice porta in generale a ridurre di molto i costi di esecuzione del Nested Loops Join

Accesso mediante indici: esempio

- Si fa il join tra Department e Employee (WorkDept = DeptNo) e si ha:
  NP(Department) = 20 NR(Department) = 100
  NP(Employee) = 1000 NR(Employee) = 10000
- Inoltre si considera la presenza dei seguenti indici:
  WDidx su Employee(WorkDept) con h(WDidx) = 1
  DNidx su Department(DeptNo) con h(DNidx) = 2

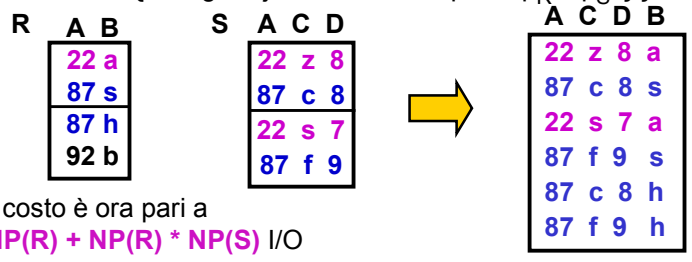
Department esterna:
NP(Department) + NR(Department) * (h(WDidx) + 10000/100) = **10120 I/O**
Employee esterna:
NP(Employee) + NR(Employee) * (h(DNidx) + 1) = **21000 I/O**

- Pertanto, con 20 msec per I/O, nel primo caso adesso sono necessari circa 202 secondi

Block Nested Loops Join

- Molti DBMS usano una variante del Nested Loops, detta Block Nested Loops, che, rinunciando a preservare l'ordine della relazione esterna, risulta più efficiente in quanto esegue il join di tutte le tuple in memoria prima di richiedere nuove pagine della relazione interna

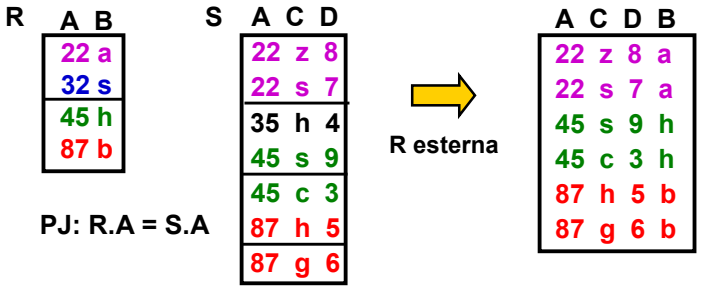
  Per ogni pagina p_R di R**:**
  { Per ogni pagina p_S in S**:**
  { esegui il join di tutte le tuple in p_R e p_S **} }**



- Il costo è ora pari a
  NP(R) + NP(R) * NP(S) I/O
- La strategia si estende anche al caso in cui a R siano assegnati più buffer

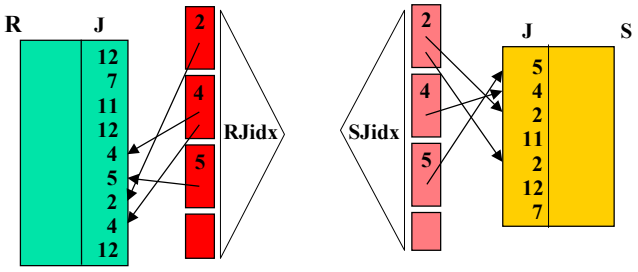
Merge Scan Join

- Il Merge Scan Join è applicabile quando entrambi gli insiemi di tuple in input sono ordinati sugli attributi di join
- Per R (S) ciò è possibile se:
  - R (S) è fisicamente ordinata sugli attributi di join
  - Esiste un indice sugli attributi di join di R (S)



Merge Scan Join: costi

- La logica dell'algoritmo sfrutta il fatto che entrambi gli input sono ordinati per evitare di fare inutili confronti, il che fa sì che il numero di letture sia dell'ordine di **NP(R) + NP(S)** se si accede sequenzialmente alle due relazioni
- Con indici unclustered il costo può arrivare a **NR(R) + NR(S)**

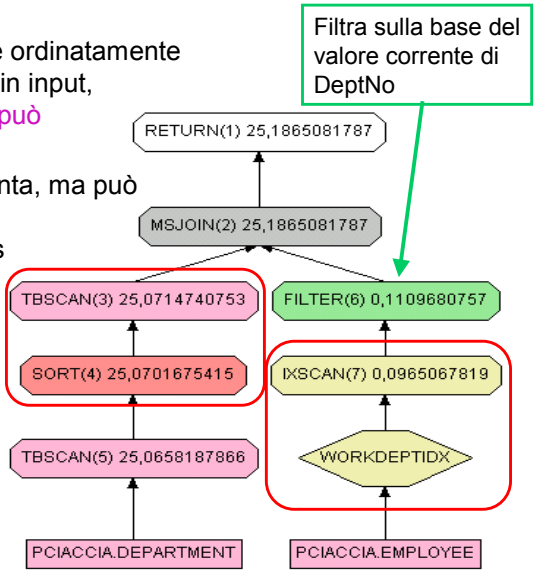


- Il Merge Scan è in generale usato solo per predicati di join di uguaglianza (equi-join), perché negli altri casi i suoi vantaggi si riducono considerevolmente

Merge Scan Join: sort degli input

- Se non è possibile accedere ordinatamente a uno o entrambi gli insiemi in input, l'ottimizzatore considera se può convenire eseguire il Sort
- Il costo di esecuzione aumenta, ma può comunque risultare minore di quello di un Nested Loops

```
SELECT WorkDept, MgrNo
FROM Department,Employee
WHERE  WorkDept = DeptNo
```



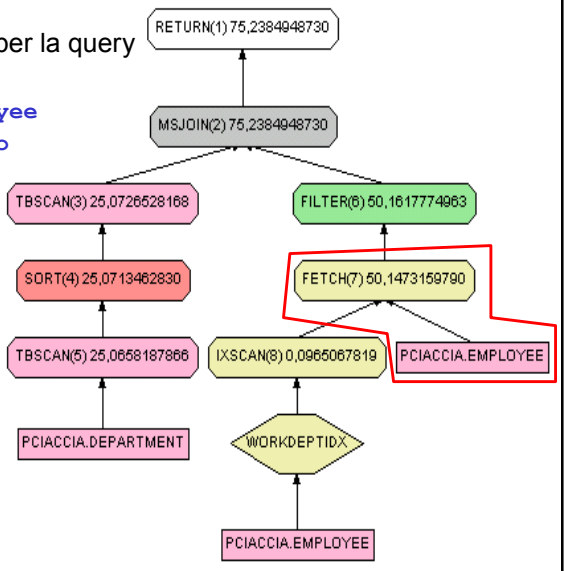
Merge Scan Join: esempio

- Il piano di accesso in figura per la query

```
SELECT *
FROM   Department,Employee
WHERE  WorkDept = DeptNo
```

usa Merge Sort Join

- Fa inoltre uso dell'operatore **Fetch** che, data una lista di RID, reperisce i corrispondenti record della relazione Employee



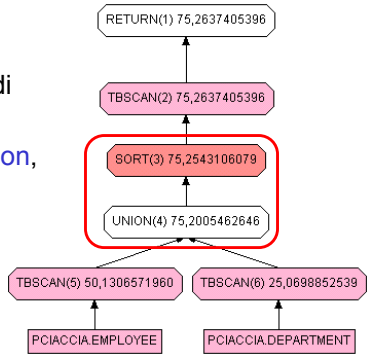
Hash Join

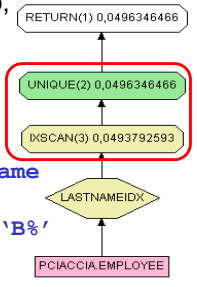
- L'algoritmo di Hash Join, applicabile solo in caso di equi-join, non richiede né la presenza di indici né input ordinati, e risulta particolarmente vantaggioso in caso di relazioni molto grandi
- L'idea su cui si basa l'Hash Join è semplice:
  - Si suppone di avere a disposizione una funzione hash H, che viene applicata agli attributi di join delle due relazioni (ad es. R.J e S.J)
  - Se t_R e t_S sono 2 tuple di R e S, allora

    è possibile che sia $t_R.J = t_S.J$ solo se $H(t_R.J) = H(t_S.J)$
  - Se, viceversa, $H(t_R.J) <> H(t_S.J)$, allora sicuramente è $t_R.J <> t_S.J$
- A partire da questa idea si hanno diverse implementazioni, che hanno in comune il fatto che R e S vengono partizionate sulla base dei valori di H, e che la ricerca di matching tuples avviene solo tra partizioni relative allo stesso valore di H
- Il costo risulta essere dell'ordine di **NP(R) + NP(S)**, ma dipende anche fortemente dal numero di buffer a disposizione e dalla distribuzione dei valori degli attributi di join (il caso uniforme è quello migliore)

Altri operatori

- Ogni DBMS ha a disposizione un vasto repertorio di operatori, ma molti di questi di fatto sono varianti di (o usano) Sort e Join
- Ad esempio, in DB2 esiste l'operatore Union, che esegue l'unione senza eliminare i duplicati; per eliminarli si usa il Sort che li rimuove durante l'ordinamento
- Se l'input è già ordinato, per eliminare i duplicati DB2 fa uso dell'operatore Unique






```
SELECT DISTINCT LastName
FROM   Employee
WHERE  LastName LIKE 'B%'
```

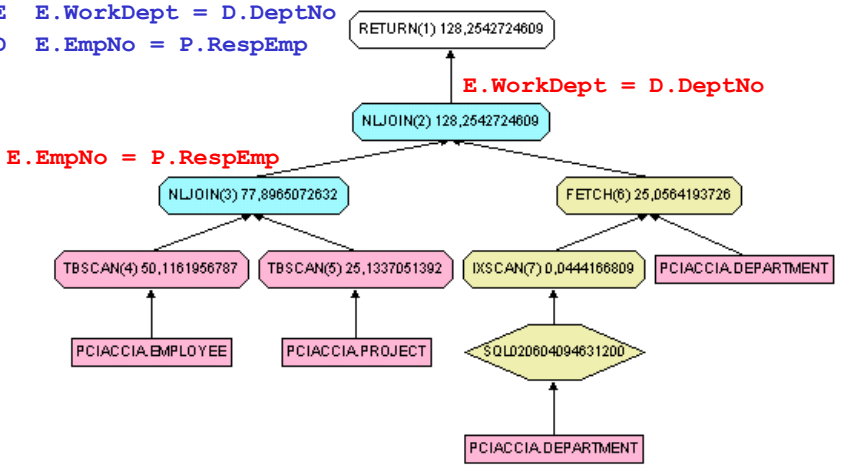
```
SELECT MgrNo
FROM Department
UNION
SELECT EmpNo
FROM Employee
WHERE HireDate >'31/12/2001'
```

Valutazione per materializzazione

- Un semplice modo di eseguire un piano di accesso composto da diversi operatori consiste nel procedere bottom-up, secondo il seguente schema:
  - Si calcolano innanzitutto i risultati degli operatori al livello più basso dell'albero e si memorizzano tali risultati in relazioni temporanee
  - Si procede quindi in modo analogo per gli operatori del livello sovrastante, fino ad arrivare alla radice
- Tale modo di procedere, detto "**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

Materializzazione: esempio (1)

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



Materializzazione: esempio (2)

- Sull'istanza del DB Sample, la valutazione per materializzazione produrrebbe come risultato del primo Join (`E.EmpNo = P.RespEmp`):

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 Join `E.WorkDept = D.DeptNo`

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	

Valutazione 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 precedente, la valutazione in pipeline opererebbe così:
  - Si inizia a eseguire il primo Join (`e.EmpNo = p.RespEmp`). Appena viene prodotta la prima tupla dell'output: 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:

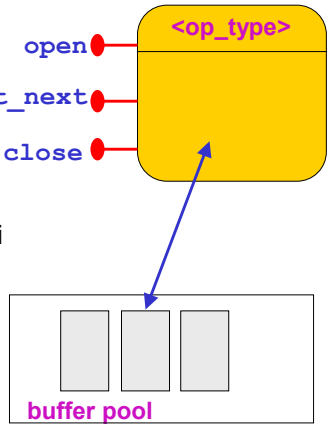
PROJNO	EMPNO	WORKDEPT
AD3100	000010	A00

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

  - La valutazione 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 Join di produrre un'altra tupla

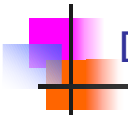
Interfaccia a iteratore

- Per supportare una valutazione in pipeline e per semplificare la realizzazione dei diversi operatori, i DBMS definiscono gli operatori mediante un'interfaccia a "iteratore", i cui metodi principali sono:
  - `open`: inizializza lo stato dell'operatore, alloca buffer per gli input e l'output, richiama ricorsivamente `open` sugli operatori figli; viene anche usato per passare argomenti (ad es. la condizione che un operatore Filter deve applicare)
  - `get_next`: usato per richiedere un'altra tupla del risultato dell'operatore; l'implementazione di questo metodo include `get_next` sugli operatori figli e codice specifico dell'operatore
  - `close`: usato per terminare l'esecuzione dell'operatore, con conseguente rilascio delle risorse ad esso allocate



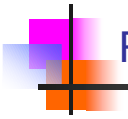
Pipeline con iteratori

- L'interfaccia a iteratore supporta naturalmente una 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 metodi di accesso quali i B^+-tree, che esternamente vengono quindi visti semplicemente come operatori che producono un insieme (stream) di tuple



Determinazione del piano ottimale

- L'ottimizzatore, al fine di poter determinare il piano di accesso a costo minimo, dispone di una strategia di enumerazione (generazione) dei piani di accesso, i cui compiti principali sono:
  - Enumerare tutti i piani che, potenzialmente, possono risultare ottimali
  - Non generare piani di accesso che sicuramente non possono risultare ottimali
- Il numero di piani di accesso che vengono generati per ottimizzare una query può risultare molto elevato
  - Ad esempio, se una query esegue il join di N relazioni, esistono almeno N! piani di accesso potenzialmente ottimali che l'ottimizzatore deve considerare
- Diversi DBMS permettono di controllare esplicitamente la strategia di enumerazione, in modo da permettere un "tuning" più fine delle prestazioni



Riassumiamo:

- L'elaborazione di una query SQL comporta l'esecuzione di una serie di passi, in particolare parsing, check semantico, riscrittura e ottimizzazione basata sui costi
- La fase di check semantico, oltre a verificare l'esistenza degli oggetti referenziati dalla query, controlla anche che l'utente abbia i privilegi necessari per eseguire l'operazione. A tale scopo si fa riferimento ai cataloghi, in cui sono memorizzate tutte le informazioni necessarie
- Nella fase di riscrittura vengono risolti i riferimenti alle viste, si cerca di eseguire l'unnesting di subquery e si semplifica la query facendo uso dei vincoli definiti a livello di schema
- La fase di ottimizzazione basata sui costi fa uso intensivo delle statistiche sui dati (memorizzate nei cataloghi) per determinare il piano di accesso a costo minimo, componendo in modo opportuno gli operatori fisici a disposizione
- Il piano così generato viene eseguito sfruttando l'interfaccia a iteratore degli operatori, che supporta naturalmente una modalità di esecuzione in pipeline