NUMERO: 1469A - ANNO: 2015

A P P U N T I

STUDENTE: De Donno

MATERIA: Sistemi per la Gestione di Basi di Dati. Prof.Baralis

Sistemi per la Gestione di Basi di Dati

(Oracle)

A cura di ***Michele De Donno***

© Proprietà riservata dell'autore - Digitalizzazione e distribuzione a cura del CENTRO APPUNTI - Corso Luigi Einaudi, 55 - Torino / Pagina 3 di 598

L'obiettivo è quello di determinare se è vantaggioso **modificare** la forma della query in modo da consentire la generazione di un miglior piano di esecuzione per la query stessa.

Estimatore

L'obiettivo dell'estimatore è quello di **stimare** il costo complessivo di un determinato piano di esecuzione, sfruttando tre diversi tipi di misure:

- **Selettività** - rappresenta una **frazione** di righe rispetto ad un insieme di righe;
- **Cardinalità** - rappresenta il **numero** di righe rispetto ad un insieme di righe;
- **Costo** - rappresenta le unità di **lavoro** o le **risorse** utilizzate. L'ottimizzatore delle query utilizza come unità di lavoro il numero di *I / O* da disco (metodo di accesso), l'utilizzo della *CPU* e l'utilizzo della *memoria*.

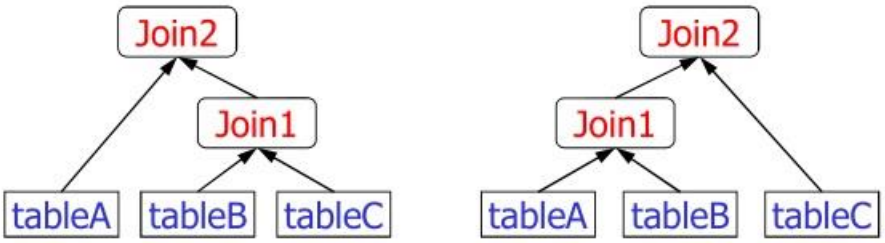
L'estimatore utilizza inoltre **statistiche** contenute nel dizionario dei dati per calcolare delle misure che migliorano il grado di accuratezza della valutazione: **istogramma** dei diversi valori presenti in una colonna di una tabella.

Generatore dei piani di esecuzione

La funzione principale del **generatore dei piani** è quella di analizzare **tutti i** diversi **possibili** piani di esecuzione per una data query e determinare quello avente **costo minore**.

Sono possibili molti piani di esecuzione per via della combinazione di differenti:

- Metodi di accesso ai dati;
- Metodi di join;
- Ordini di join.



Tale blocco utilizza un **limite** interno per ridurre il numero di piani analizzati; tale limite è basato sul costo di esecuzione del **miglior piano** trovato fino a quel momento: se il limite è alto sono esplorati più piani di esecuzione, se invece è basso saranno di meno.

Esempio

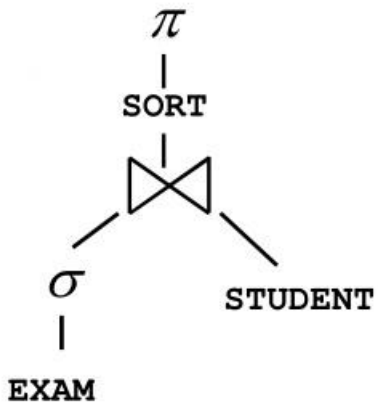
Si consideri una base dati costituita dalle seguenti tabelle:

STUDENT (SId, SSurname, SName)
COURSE (CCode, PId, Year, Semester)
EXAM (CCode, SId, Date, Score)

E si supponga di dover eseguire la seguente query: "Indicare nome e matricola degli studenti che hanno superato almeno un esame con un voto maggiore o uguale a 27".

```
SELECT SName, S.Sid
FROM EXAM E, STUDENT S
WHERE S.Sid=E.Sid and Score>=27
ORDER BY SName
```

- *1° passo* – Calcolare l'**espressione algebrica** associata alla query SQL:



Le operazioni da eseguire saranno quindi, leggendo dal basso verso l'alto, le seguenti:

1. Leggere i dati memorizzati nelle due tabelle: **EXAM**, **STUDENT**
2. Filtrare i record da leggere in funzione della condizione specificata nella clausola `WHERE`: **σ**
3. Eseguire il join tra le due tabelle: ⋈
4. Effettuare l'ordinamento rispetto agli attributi specificati nella clausola `ORDER BY`: **SORT**
5. Effettuare la proiezione rispetto agli attributi indicati nella clausola `SELECT`: **π**

N.B. L'operatore `ORDER BY` viene sempre eseguito come ultima operazione prima di mostrare il risultato.

La tabella relativa al piano di esecuzione contiene anche informazioni riguardanti i seguenti argomenti:

- **Ottimizzazione** – Ad esempio il costo e la cardinalità di ogni operazione
- **Partizionamento** – Ad esempio l'insieme delle partizioni accedute
- **Esecuzione parallela** – Ad esempio il metodo di distribuzione dell'ordine in ingresso al join.

Capire l'Ottimizzatore delle Query

Il Query Optimizer determina, per una data istruzione SQL, quale piano di esecuzione è più efficiente (cioè, ha il costo più basso), tramite le seguenti azioni:

- Prendendo in considerazione i **metodi di accesso** disponibili;
- Cambiando **l'ordine** di esecuzione dei **join**;
- Valutando diversi **metodi di join**;
- Analizzando le **statistiche** prese dal dizionario dati relative agli oggetti dello schema (tabelle o indici) a cui si accede nell'istruzione SQL.

Metodi di Accesso per il Query Optimizer

I metodi di accesso consentono il **recupero dei dati** dal database. In particolare i metodi di accesso possono essere divisi in due macro-categorie:

- Accesso tramite **indice** – I metodi di accesso con indice vengono utilizzati per le istruzioni che accedono ad un piccolo sottoinsieme delle tuple della tabella.
  L'accesso tramite indice può inoltre essere:
  - *Esaustivo*: è sufficiente leggere *solo* il contenuto dell'indice per rispondere alla query.
  - *Parziale*: è necessario prima leggere i dati memorizzati nell'indice e successivamente accedere fisicamente alle tuple della tabella a cui si riferiscono (Es. indice è definito su uno solo degli attributi interessati dalla query per una determinata tabella).
- Accesso direttamente alla **tabella** – Si effettua la scansione completa della tabella quando si deve accedere ad una grande porzione dei dati contenuti nella tabella stessa .

In Oracle, i dati possono essere recuperati da qualsiasi tabella tramite i seguenti metodi di accesso:

- **Full Table Scans** – Accesso direttamente alla tabella.
- **Index Scans** – Accessi esaustivo tramite indice.
- **Rowid Scans** – Accesso parziale tramite indice.

Valutazione di I/O a blocchi

Oracle esegue le operazioni di I/O a **blocchi**.

Generalmente **più righe** vengono memorizzate in ciascun blocco. Il numero totale delle righe di una tabella possono essere raggruppate in pochi blocchi, o potrebbero essere sparse su un gran numero di blocchi.

La decisione dell'ottimizzatore di utilizzare un full table scan è influenzata dalla **percentuale di blocchi** ai quali è necessario accedere, e non dal numero di righe. Questo è chiamato **Index Clustering Factor**, ed è un valore che fornisce una indicazione di come i dati sono distribuiti sui diversi blocchi della tabella.

Anche se l'**index clustering factor** è una proprietà dell'indice, in realtà si riferisce alla diffusione dei **valori simili delle colonne indicizzate** all'interno dei blocchi dati che compongono la tabella:

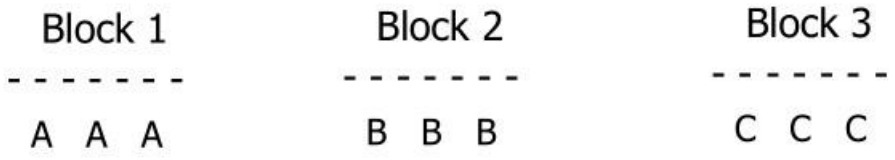
- Index clustering factor **Basso**: le singole righe sono **concentrate** in pochi blocchi della tabella.
- Index clustering factor **Alto**: le singole righe sono **sparse** in maniera casuale tra i blocchi della tabella.
  In questo caso è più costoso utilizzare una scansione ad intervallo per andare a prelevare le righe in funzione del rowid, in quanto dovranno essere letti più blocchi della tabella per ottenere i dati voluti.

Effetto del Clustering Factor sul costo

Si consideri la seguente situazione:

- Vi è una tabella di **9 righe**.
- Vi è un indice non-unique definito sulla **colonna 1**.
- La colonna 1 contiene i valori **A**, **B** e **C**.
- Oracle memorizzate la tabella tramite **3 blocchi**.

Caso 1. I dati relativi ad ogni valore assunto dalla colonna 1 sono memorizzati nello stesso blocco. In questo caso l'index Clustering Factor è **basso** e le righe sono organizzate come mostrato nella figura successiva:



Quando l'index clustering factor è basso *non è conveniente* eseguire un full table scan.

© Proprietà riservata dell'autore - Digitalizzazione e distribuzione a cura del CENTRO APPUNTI - Corso Luigi Einaudi, 55 - Torino / Pagina 11 di 598

Indici **secondari**:

- Btree
- Bitmap
- Hash

Per **creare** un indice in Oracle si utilizza la seguente sintassi:

```
CREATE INDEX IndexName ON Table (Column, …);
```

Si osservi che Oracle non permette di definire il tipo di indice creato (Hash, B-Tree), è il sistema a deciderlo (generalmente B^+-Tree).

Per **eliminare** un indice in Oracle si utilizza la seguente sintassi:

```
DROP INDEX IndexName;
```

Scansioni tramite indice

L'indice contiene il **valore** indicizzato e i **rowid** (indirizzo fisico di ogni record) relativi alle righe della tabella che assumono tale valore.

Una scansione tramite indice recupera i dati da un indice in funzione del valore di una o più colonne dell'indice stesso:

- Oracle cerca l'indice per i **valori** delle colonne indicizzate a cui si **accede nell'istruzione**.
- Se l'istruzione accede solo alle colonne dell'indice, i valori della colonna indicizzata vengono letti **direttamente dall'indice**, altrimenti si accede alle righe della tabella tramite il **rowid**.

Le scansioni tramite indice possono essere di diversi tipi:

- Index Unique Scans
- Index Range Scans
- Index Full Scans
- Fast Full Index Scans
- Bitmap Indexes

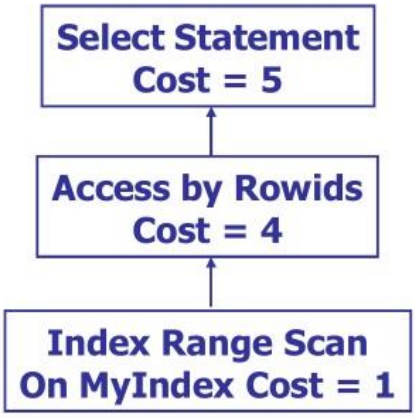
Il range scan può essere eseguito su indici unique o non-unique. Tale tipo di operazione permette di **evitare l'ordinamento** delle tabelle quando la colonna dell'indice costituisce l'attributo sul quale è stata definita una clausola di `ORDER BY`/`GROUP BY`.

Esempio

```
STUDENT (SId, SSurname, SName)
COURSE (CCode, PCode, Year, Semester)
EXAM (CCode, SId, Date, Score)
```

```
Query: SELECT SId, CCode, Score
       FROM EXAM
       WHERE Score>=27;
```

```
CREATE INDEX MyIndex On EXAM(Score);
```



Index Full Scans

L'index full scan è il metodo che viene utilizzato per leggere tutto il contenuto di un indice.

Una scansione di tipo index full scan è disponibile se un predicato fa riferimento a una delle **colonne** dell'indice. Il predicato non deve essere necessariamente un index driver.

Tale tipo di scansione è disponibile anche quando **non** vi è un **predicato**, solo se entrambe le seguenti condizioni sono soddisfatte:

- **Tutte** le **colonne** della tabella a cui si fa riferimento nella query sono incluse nell'indice.
- Almeno una delle colonne dell'indice ha un vincolo di **`not null`**.

Una full scan può essere utilizzata per eliminare un'operazione di ordinamento (richiesta dalla `GROUP BY`, `ORDER BY`, `MERGE JOIN`), in quanto i dati sono **ordinati** rispetto alla chiave dell'indice.

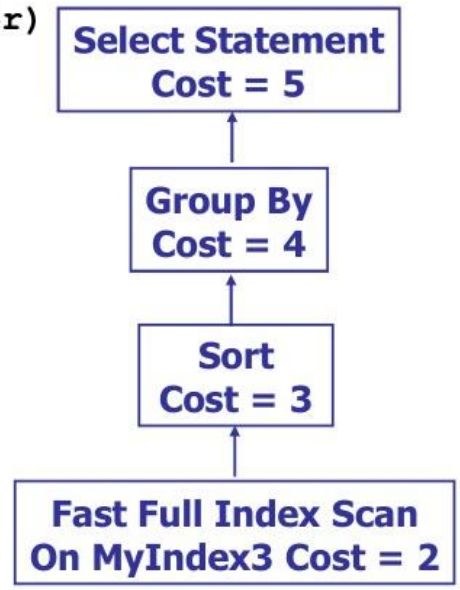
In una operazione di questo tipo, i blocchi sono letti singolarmente (uno per volta).

Esempio

```
STUDENT (SId, SSurname, SName)
COURSE (CCode, PCode, Year, Semester)
EXAM (CCode, SId, Date, Score)

Query: SELECT CCode, AVG(Score)
       FROM EXAM
       GROUP BY CCode;

  CREATE INDEX MyIndex3
  On EXAM(CCode,Score);
```



Bitmap Indexes

Il metodo Bitmap index può essere utilizzato solo quando sono presenti degli indici bitmap sulla tabella ed è molto efficiente per query che contengono **condizioni multiple**, relative alla stessa tabella, nella clausola `WHERE`.

Gli indici bitmap sono generalmente più facili da distruggere e ricostruire piuttosto che da mantenere, per questo motivo vengono tipicamente creati "**al volo**" dal sistema quando necessario.

Un **`JOIN BITMAP`** utilizza un indice bitmap per i valori chiave delle tabelle e una funzione di mapping che converte ogni posizione di bit nel corrispondente rowid.

Inoltre, un indice Bitmap può unire efficacemente gli indici che corrispondono a diverse condizioni nella clausola WHERE, utilizzando **operazioni booleane** per risolvere condizioni di `AND` o `OR`.

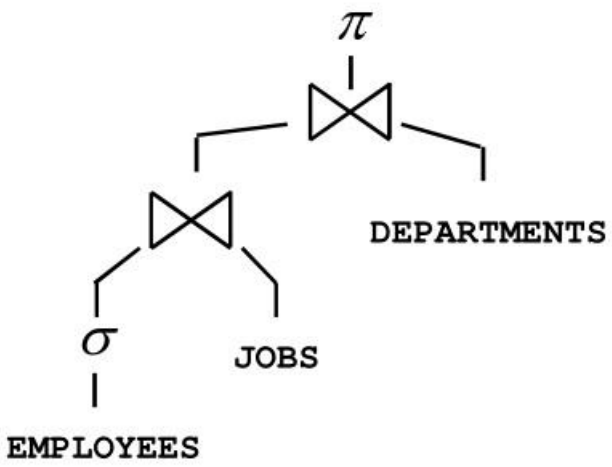
Esempio

Si consideri il seguente schema e la seguente query definita su di esso:

```
EMPLOYEES(
        employee_id,
        department_id,
        job_id, name, birth_date, salary)
JOBS( job_id,
        grade, job_title, name)
DEPARTMENTS(
        department_id,
        department_name, city)
```

```
EXPLAIN PLAN FOR
SELECT e.employee_id, j.job_title, e.salary, d.department_name
    FROM employees e, jobs j, departments d
    WHERE  e.employee_id < 103
       AND e.job_id = j.job_id
       AND e.department_id = d.department_id;
```

L'espressione algebrica associata alla query sarà la seguente:



L'ordine di join scelto è arbitrario.

Il piano di esecuzione di Oracle per tale query è il seguente:

- Accesso tramite **index unique scan** alla tabella *DEPARTMENTS* – questo perché l'indice è definito sulla chiave primaria e quindi ad ogni valore dell'indice corrisponderà una sola tupla.
    - L'indice non è coprente in quanto è definito solo sulla chiave primaria ma è necessario accedere anche all'attributo *department_name*, per cui l'accesso fisico alla tabella è necessario.

Join

Metodi di Join

Per unire ogni **coppia di righe**, Oracle deve eseguire una operazione di join. I metodi di join includono:

- Nested Loop
- Sort Merge
- Hash Joins

La versione attuale di Oracle tende a scegliere sempre l'hash join in quanto è la modalità di join implementata in modo più efficiente.

Ordine di join

Per eseguire un'istruzione che esegua il **join di più di due tabelle**, Oracle esegue il join di due di esse e successivamente esegue il join del risultato con la tabella successiva. Questo processo continua finché non è stato eseguito il join di tutte le tabelle.

Nested Loop Joins

I **nested loop join** sono utili quando **piccoli sottoinsiemi di dati** devono essere messi in join tra solo e la condizione di join costituisce un modo efficiente di **accedere alla seconda tabella**.

Un join eseguito tramite tecnica nested loop si realizza tramite i seguenti passi:

- L'ottimizzatore determina qual è la tabella principale e la identifica come **outer table**.
- L'altra tabella interessata nel join viene identificata come **inner table**.
- Per **ogni riga** della *outer table*, Oracle accede a **tutte le righe** della *inner table*.

L'**outer loop** si esegue per ogni riga della outer table e l'**inner loop** per ogni riga della inner table. L'outer loop compare prima dell'inner loop all'interno del piano di esecuzione.

Hash Joins

Gli hash join sono utilizzati per eseguire il join tra **grandi insiemi di dati**. L'ottimizzatore usa la **più piccola** delle tabelle o sorgenti di dati per costruire in memoria una **hash table** rispetto al valore dell'attributo di join. Successivamente esso scansiona la tabella più grande effettuando il **probing** (confronto, sondaggio) con l'hash table per cercare le righe sulle quali effettuare il join.

Questo metodo è tipicamente utilizzato quando la tabella più piccola interessata dal join può essere tenuta **completamente nella memoria** disponibile. In questo caso il costo è infatti limitato ad una singola lettura dei dati delle due tabelle.

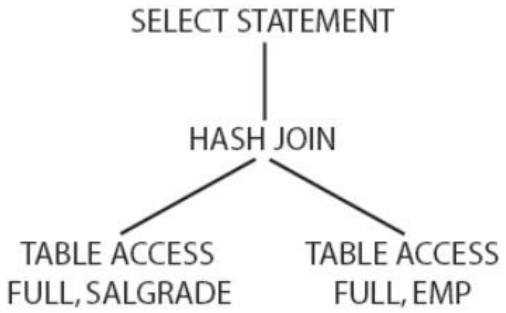
L'ottimizzatore usa un hash join per unire due tabelle per le quali deve essere eseguito un **equijoin** e se una delle seguenti condizioni è vera:

- Deve essere eseguito il join di una grande **quantità** di dati.
- Una grande **porzione** di una piccola tabella è interessata dal join.

Esempio

```
EMP( Empno, Ename, Job, Mgr,
  Hiredate, Sal, Comm, Deptno)
DEPT( Deptno, Dname, Loc)
SALGRADE( Grade, Losal, Hisal)
```

```
SELECT *
FROM EMP E, SALGRADE S
WHERE E.Sal = S.Losal
      AND E.Job = 'RESEARCHER';
```



Sort Merge Joins

Il sort merge join può essere utilizzato per eseguire il join di righe provenienti da due sorgenti dati indipendenti. Questo tipo di join risulta più performante rispetto all'**hash join** se le condizioni seguenti sono soddisfatte:

- Le righe sorgenti sono **già ordinate**;
- **Non** deve essere eseguita una operazione di **ordinamento** (Es. dopo un GROUP BY).
- Una operazioni di **ordinamento** può essere eseguita per l'**operazione successiva** (Es. prima di un GROUP BY).

Il sort merge join è utile quando la condizione di join tra le due tabelle è una condizione di **disuguaglianza** (ma non una condizione di non-uguaglianza: <>) come <, ≤, ≥ o >.

© Proprietà riservata dell'autore - Digitalizzazione e distribuzione a cura del CENTRO APPUNTI - Corso Luigi Einaudi, 55 - Torino / Pagina 23 di 598

Statistiche di **attributo**:

- Numero di valori distinti di ogni attributo (NDV)
- Numero di valori null di ogni attributo
- Distribuzione dei dati (istogramma)

Statistiche di **indice**:

- Numero di blocchi foglia
- Livelli
- Clustering factor

Statistiche di **sistema**:

- Utilizzo e prestazione delle operazioni I/O
- Utilizzo e prestazione delle operazioni di CPU

Per visualizzare le statistiche presenti nel dizionario dati è sufficiente **interrogare** l'opportuna **vista** materializzata (*USER, ALL o DBA*) in esso contenuta. In particolare:

- Viste DBA – possono essere interrogate solo dall'amministratore del database;
- Viste USER – possono essere interrogate dagli utenti del DBMS;
- Viste ALL – possono essere interrogate da chiunque;

Le viste materializzate DBA sono le seguenti:

- DBA_**TABLES**
- DBA_OBJECT_TABLES
- DBA_**TAB_STATISTICS**
- DBA_**TAB_COL_STATISTICS**
- DBA_**TAB_HISTOGRAMS**
- DBA_**INDEXES**
- DBA_IND_STATISTICS
- DBA_CLUSTERS
- DBA_TAB_PARTITIONS
- DBA_TAB_SUBPARTITIONS
- DBA_IND_PARTITIONS
- DBA_IND_SUBPARTITIONS
- DBA_PART_COL_STATISTICS
- DBA_PART_HISTOGRAMS
- DBA_SUBPART_COL_STATISTICS
- DBA_SUBPART_HISTOGRAMS

Le viste di tipo ALL e USER disponibili sono le stesse viste per DBA, è sufficiente sostituire il prefisso (DBA) in modo opportuno (ALL o USER).

© Proprietà riservata dell'autore - Digitalizzazione e distribuzione a cura del CENTRO APPUNTI - Corso Luigi Einaudi, 55 - Torino / Pagina 25 di 598

La frequenza degli intervalli di raccolta delle statistiche deve, da un lato, permettere di fornire statistiche **accurate** all'ottimizzatore e, dall'altro, tenere in considerazione il **carico di elaborazione** necessario per la raccolta di tali statistiche.

Statistiche di Attributo ed Istogrammi

Quando vengono raccolte statistiche su una tabella, il **DBMS_STATS** raccoglie informazioni relative alla distribuzione dei dati degli attributi nella tabella (Es. il valore massimo e minimo di un attributo)

Per distribuzioni di dati non regolari (skewed), possono essere creati degli **istogrammi** come parte delle statistiche di attributo al fine di descrivere la distribuzione dei dati di tale attributo.

Istogrammi

Le statistiche di attributo possono essere memorizzate sotto forma di **istogrammi** i quali forniscono delle stime **accurate** della distribuzione dei dati dell'attributo.

Gli istogrammi consentono una migliore stima di **selettività** in presenza di dati non regolari (skew), risultante in piani di esecuzione ottimali con distribuzioni di dati **non uniformi**.

Oracle usa due tipi di istogrammi per le statistiche di attributo:

- **Height-balanced histograms**
- **Frequency histograms**

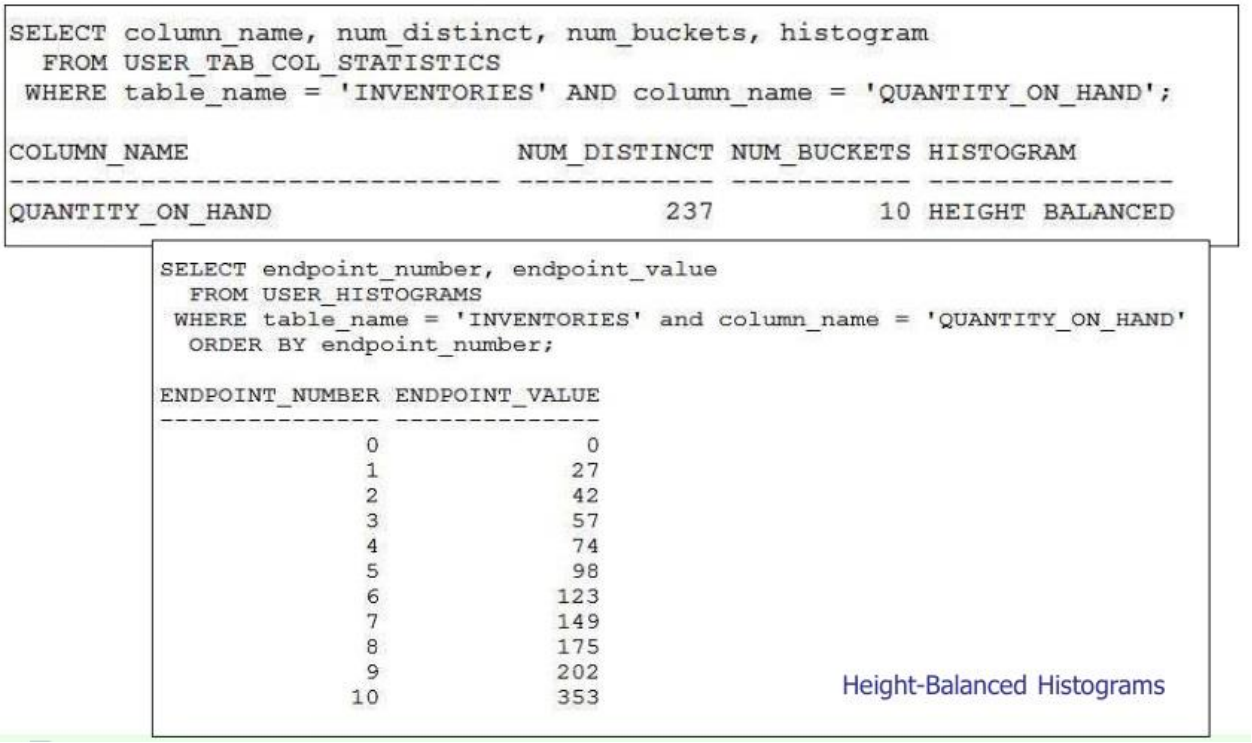
Le informazioni relative agli istogrammi sono memorizzate nella vista materializzata **USER_TAB_COL_STATISTICS**. In particolare, il tipo di istogramma viene memorizzato nella colonna **HISTOGRAM** delle viste USER/DBA_TAB_COL_STATISTICS.

Nella vista materializzata **USER_HISTOGRAMS** sono contenute le informazioni relative a come i dati sono distribuiti.

Height-Balanced Histograms

In un istogramma height-balanced, i valori delle colonne sono divisi in gruppi, in modo che ognuno di essi contenga approssimativamente lo **stesso numero di righe**. Ogni blocco (bucket) è identificato da un ID (*Endpoint_Number*) ed è caratterizzato da un valore (*Endpoint_Value*) che rappresenta il valore più grande presente all'interno del blocco.

L'informazione utile che fornisce tale tipo di istogramma è quella di capire dove cadono gli endpoint all'interno del range di valori. In pratica, data una query ad intervallo, permette di capire la percentuale di righe che soddisfano il predicato di selezione e quindi la **selettività** del predicato, semplicemente guardando i valori massimi contenuti nell'istogramma.



```
SELECT column_name, num_distinct, num_buckets, histogram
  FROM USER_TAB_COL_STATISTICS
 WHERE table_name = 'INVENTORIES' AND column_name = 'QUANTITY_ON_HAND';

COLUMN_NAME                    NUM_DISTINCT NUM_BUCKETS HISTOGRAM
------------------------------ ------------ ----------- ---------------
QUANTITY_ON_HAND                        237          10 HEIGHT BALANCED
```

```
SELECT endpoint_number, endpoint_value
  FROM USER_HISTOGRAMS
 WHERE table_name = 'INVENTORIES' and column_name = 'QUANTITY_ON_HAND'
  ORDER BY endpoint_number;

ENDPOINT_NUMBER ENDPOINT_VALUE
--------------- --------------
              0              0
              1             27
              2             42
              3             57
              4             74
              5             98
              6            123
              7            149
              8            175
              9            202
             10            353
```

Height-Balanced Histograms

Dai dati mostrati in figura appare evidente che se vi è una query con un predicato di selezione del tipo "$Quantity\_on\_hand < 200$" tale query selezionerà il 90% (9 bucket su 10) delle tuple della tabella *INVENTORIES*.

Frequency Histograms

In un istogramma di frequenza, **ogni valore** dell'attributo corrisponde ad un singolo **bucket** dell'istogramma. Ogni blocco (bucket) contiene il numero di **occorrenze** di quel singolo valore.

Gli istogrammi di frequenza sono creati automaticamente (contrariamente agli istogrammi Height-Balanced) quando il numero di **valori distinti** di un attributo è minore o uguale al numero di **blocchi** specificati per l'istogramma.

Anche le informazioni relative gli istogrammi di frequenza possono essere visualizzate usando le tabelle **USER_HISTOGRAMS** e **USER_TAB_COL_STATISTICS:**

- NUM_DISTINCT – numero di valori distinti assunti dall'attributo;
- NUM_BUCKETS – numero di buckets definiti per l'attributo;
- HISTOGRAM – Tipo di istogramma definito.

- ***ENDPOINT_NUMBER*** – frequenza *cumulativa* di ogni valore all'interno dell'attributo: la frequenza del valore i-esimo assunto dall'attributo è data da:

$$ENDPOINT\_NUMBER_i - ENDPOINT\_NUMBER_{i-1}$$

Ad esempio il valore 3 è assunto dall'attributo $261 - 213 = \mathbf{48}$ volte.

© Proprietà riservata dell'autore - Digitalizzazione e distribuzione a cura del CENTRO APPUNTI - Corso Luigi Einaudi, 55 - Torino / Pagina 29 di 598

Scegliere un obiettivo di ottimizzazione

*Ottimizzazione per ottenere il miglior **throughput*** – si minimizza il carico di lavoro complessivo associato alla query.

> L'ottimizzatore sceglie il minor quantitativo di risorse necessarie a processare **tutte le righe** accedute dall'istruzione SQL.
>
> Il throughput è più importante nelle **applicazioni batch** [**off-line**](Es. Oracle Reports applications) nelle quali l'utente è interessato esclusivamente al tempo necessario all'esecuzione completa dell'applicazione.

*Ottimizzazione per ottenere il miglior **tempo di risposta***

> L'ottimizzatore usa il minor numero di risorse necessarie a processare la **prima riga** acceduta dall'istruzione SQL.
>
> Il tempo di riposta è importante nelle applicazioni **interattive** [**real-time**] (Es. SQL *Plus queries).

In funzione dell'obiettivo di ottimizzazione, le scelte fatte dall'ottimizzatore cambiano. Per esempio, per quanto riguarda il metodo di join:

- Ottimizzazione del throughput → Hash Join.
- Ottimizzazione del tempo di risposta → Nested loop.

Valori dei parametri `OPTIMIZER_MODE`

Il seguente comando SQL permette di cambiare l'obiettivo di ottimizzazione del query optimizer:

`OPTIMIZER_MODE =` *value*

Al posto di *value* possono essere inseriti i seguenti parametri:

- `ALL_ROWS` – L'ottimizzatore usa un approccio **basato sui costi** per tutte le istruzioni SQL della sessione. L'obiettivo di ottimizzazione è di migliorare il **throughput** (minor quantitativo di risorse necessarie ad eseguire l'intera istruzione SQL). **Default**.
- `FIRST_ROW_n` - L'ottimizzatore usa un approccio **basato sui costi** con l'obiettivo di fornire il miglior **tempo di riposta** necessario a restituire le prime ***n*** righe del risultato; dove *n* può essere uguale a 1, 10, 100 o 1000.
- `FIRST_ROW` - L'ottimizzatore usa un approccio misto **basato su costi ed euristiche** al fine di trovare il miglior piano di esecuzione che fornisca velocemente le prime righe del risultato.

Come svolgere esercizi di ottimizzazione fisica

Data una query:

1. *Scrivere l'espressione algebrica associata alla query;*
2. *Scrivere il metodo di accesso ai dati utilizzato dall'ottimizzatore per eseguire ogni operazione (in assenza di strutture fisiche accessorie);*
3. *Valutare i metodi di join da utilizzare;*
4. *Definire eventuali strutture fisiche accessorie che possono essere utilizzate per migliorare il piano di esecuzione (ridurre il costo di esecuzione della query);*
5. *Scrivere il nuovo piano di esecuzione che utilizzi le strutture fisiche definite.*

Categorie di hint

I suggerimenti che possono essere forniti all'ottimizzatore sono raggruppati nelle seguenti categorie:

- Hint per gli **Approcci e Obiettivi di Ottimizzazione**;
- Hint per i **Metodi di Accesso** ai dati;
- Hint per le trasformazioni delle query;
- Hint per gli **Ordini di Join**;
- Hint per le **Operazioni di Join**;
- Hint per l'esecuzione parallela;
- Hint addizionali.

Strategie e Obiettivi di Ottimizzazione

I seguenti suggerimenti consentono di scegliere tra diversi approcci e obiettivi di ottimizzazione:

- `ALL_ROWS` – ottimizza un blocco di istruzione con l'obiettivo di migliorare il **throughput** (minor consumo totale di risorse).
- `FIRST_ROWS(n)` – ottimizza un'istruzione SQL per ridurne il **tempo di risposta**, scegliendo il piano di esecuzione che restituisce le prime *n* righe nel modo più efficiente possibile.

Se un'istruzione SQL ha un suggerimento che specifica un approccio e obiettivo di ottimizzazione, l'ottimizzatore utilizzerà l'approccio specificato, indipendentemente dalla presenza o assenza di:

- Statistiche (se assenti, l'ottimizzatore utilizza i valori statistici di default)
- Parametro di inizializzazione di `OPTIMIZER_MODE`
- Parametro di `OPTIMIZER_MODE` dell'istruzione `ALTER SESSION`

L'ottimizzatore dà la precedenza ai suggerimenti relativi ai **metodi di accesso** o alle **operazioni di join**, prima di `ALL_ROWS` o `FIRST_ROWS(n)`.

Oracle utilizza questi suggerimenti quando la tabella referenziata è forzata ad essere la inner table di un join; i suggerimenti sono ignorati se la tabella referenziata è la outer table.

Ordini di join

I seguenti suggerimenti indicano l'ordine di join:

- `ORDERED` – indica ad Oracle di eseguire il join delle tabelle nell'ordine con il quale appaiono nella clausola **`FROM`**.
- `LEADING(table1 table2 … )` – indica all'ottimizzatore di utilizzare l'insieme di tabelle specificato come parametro dell'hint.

Questi suggerimenti permettono di scegliere la *inner table* e la *outer table*:

- La **prima** tabella è la **outer table**;
- La **seconda** tabella è la **inner table**.

Esempio #1

```
SELECT /*+ ORDERED */ *          LEADING( e d )
FROM emp e, dept d
WHERE d.deptno = e.deptno
|   1 |  NESTED LOOPS    |         | 50012 |  3125K|   168  (48)| 00:00:03 |
|   2 |   ACCESS FULL    | EMP     | 50111 |  2202K|    88   (4)| 00:00:02 |
|   3 |   BY INDEX ROWID| DEPT    |     1 |    19 |     1   (0)| 00:00:01 |
|*  4 |    INDEX UNIQUE | SYS_... |     1 |       |     0   (0)| 00:00:01 |
```

- EMP è la outer table;
- DEPT è la inner table.

Esempio #2

```
SELECT /*+ ORDERED */ *          LEADING( d e )
FROM dept d, emp e
WHERE d.deptno = e.deptno
|   1 |  NESTED LOOPS       |         | 50012 |  3125K| 43855   (4)| 00:08:47 |
|   2 |   TABLE ACCESS FULL| DEPT    |   507 |  9633 |     3   (0)| 00:00:01 |
|*  3 |   TABLE ACCESS FULL| EMP     |    99 |  4455 |    86   (4)| 00:00:02 |
```

- DEPT è la outer table;
- EMP è la inner table.

© Proprietà riservata dell'autore - Digitalizzazione e distribuzione a cura del CENTRO APPUNTI - Corso Luigi Einaudi, 55 - Torino / Pagina 37 di 598

Data warehouse in Oracle

Estensioni al linguaggio SQL per l'analisi dei dati presenti nei data warehouse

Funzioni OLAP disponibili

Per poter supportare l'analisi OLAP su Oracle, sono stati introdotti dei nuovi costrutti che estendono il linguaggio SQL:

- **Finestre di calcolo**: *window*
  Tale operatore viene definito nella clausola `select` e va ad operare sul *risultato* ottenuto dalla query nel quale è definito.
  N.B. *Questo significa che se nella query è presente una* `group by`*, gli attributi che potranno essere utilizzati nella finestra di calcolo sono* ***solo*** *quelli presenti nella* `group by` *stessa.*

- **Funzioni di ranking**: *rank, dense rank, ...*

- **Estensione della clausola group by**: *rollup, cube, ...*

Tabella d'esempio

La tabella di riferimento per gli esempi successivi presenta il seguente *schema*:

VENDITE (Città, Data, Importo)

Esempio di raggruppamento a livello fisico

"Selezionare per ogni data l'importo e la media dell'importo considerando la riga (data) corrente e le due righe che la precedono."

Soluzione

```
SELECT Data, Importo,
          AVG(Importo) OVER ( ORDER BY Data
                                ROWS 2 PRECEDING)
          AS MediaMobile
FROM Vendite
ORDER BY Data*;
```

*L'ordinamento del risultato finale rispetto all'attributo Data non era strettamente necessario ai fini della richiesta.

Il risultato ottenuto è il seguente:

COD_A	SUM(Q)	RankVendite
A2	300	1
A5	1100	2
A4	1300	3
A6	1300	3
A1	1900	5
A3	4500	6

Si osservi che la `order by` ha come criterio di ordinamento di *default* quello **ascendente**, per cui in questo caso la posizione '1' è associata al prodotto la cui quantità totale venduta è la minore.

Se si vuole assegnare la posizione '1' al prodotto la cui quantità totale venduta è la maggiore, è sufficiente scrivere la clausola `order by` con criterio di ordinamento **discendente**:

```
… ORDER BY SUM(Q)DESC …
```

Se si volessero **ordinare** i risultati **in funzione del ranking** definito dalla finestra di calcolo sarebbe sufficiente aggiungere la seguente clausola di ORDER BY:

```
…
ORDER BY RankVendite
```

L'etichetta associata alla finestra di calcolo può essere usata nella clausola di `order by` in quanto quest'ultima viene calcolata un attimo prima della visualizzazione, quando la finestra di calcolo è stata già risolta.

Esempio di dense ranking

La stessa richiesta vista per l'esempio precedente può essere risolta con la funzione di *dense ranking* evitando la perdita di valori di rank in presenza di "pari merito".

Soluzione

```
SELECT COD_A, SUM(Q),
              DENSE_RANK() OVER (ORDER BY SUM(Q))
              AS DenseRankVendite
FROM FAP
GROUP BY COD_A;
```

Selezione Top N nel ranking

Se si vogliono solo *i primi due articoli nel ranking* si può usare l'interrogazione che calcola il ranking come sotto-interrogazione e poi fare una selezione in base al campo di ranking: si utilizza una *table function*.

La sotto-interrogazione (*table function*) è specificata tra parentesi tonde subito dopo la `FROM` e viene utilizzata come se fosse una tabella.

```
SELECT *
FROM (SELECT COD_A, SUM(Q),
           RANK() OVER (ORDER BY SUM(Q)) AS RankVendite
      FROM FAP
      GROUP BY COD_A)
WHERE RankVendite<=2;
```

La **table function** viene gestita come una tabella temporanea *creata a **runtime*** ed *eliminata alla conclusione* dell'esecuzione della query principale. Gli attributi specificati nella `SELECT` della sotto-interrogazione costituiscono lo schema della table function e quindi sono gli unici attributi sui quali si può operare rispetto a tale tabella temporanea.

Il risultato ottenuto è il seguente:

COD_A	SUM(Q)	RankVendite
A2	300	1
A5	1100	2

Osservazione

Per risolvere questo tipo di interrogazioni è necessario utilizzare una *table function* in quanto la funzione di ranking viene eseguita immediatamente prima della visualizzazione della sotto-interrogazione, per cui non sarebbe possibile all'interno di un'unica query andare a specificare la condizione di ranking (`RankVendite<=2`) nella clausola `where`, in quanto l'attributo di ranking non fa parte della tabella interrogata quando viene eseguita la clausola stessa, ma sarà inserito nella tabella risultante.

Si andranno ora a trattare una serie di funzioni Oracle che permettono di arricchire o caratterizzare i dati visualizzati. Tali funzioni generalmente utilizzano il concetto di *partizionamento del dato all'interno della finestra di calcolo* (PARTITION BY)

Esempio CUME_DIST

"Partizionare gli articoli in base alla tipologia degli articoli ed effettuare un ordinamento nei gruppi in base al peso degli articoli. Associare ad ogni riga il rispettivo valore di CUME_DIST."

Soluzione

```
SELECT Tipo, Peso,
     CUME_DIST() OVER (PARTITION BY Tipo
                       ORDER BY Peso)
     AS CumePeso
FROM ART;
```

In questo modo si calcola la distribuzione dell'attributo "Peso" all'interno di ogni partizione fatta per "Tipo".

Il risultato ottenuto è il seguente:

Tipo	Peso	CumePeso		
Barra	12	1	(= 1/1)	Partizione 1
Ingranaggio	19	1	(= 1/1)	Partizione 2
Vite	12	.1	(= 1/10)	Partizione 3
Vite	14	.2	(= 2/10)	
Vite	16	.6	(= 6/10)	
Vite	16	.6	(= 6/10)	
Vite	16	.6	(= 6/10)	
Vite	16	.6	(= 6/10)	
Vite	17	.8	(= 8/10)	
Vite	17	.8	(= 8/10)	
Vite	18	.9	(= 9/10)	
Vite	20	1	(= 10/10)	

Una riga avente CumePeso = 0.xx (Es. 0.1) sta a significare che all'interno della partizione considerata (Es. Partizione 3), l'xx% (Es. 10%) delle righe assumono un valore minore o uguale a quello assunto dalla riga corrente (Es. 12) per l'attributo rispetto al quale è stato fatto l'ordinamento (Es. Peso).

NTILE

La funzione **NTILE(*n*)** permette di dividere ogni partizione in *n* sottogruppi (se possibile) ognuno con lo stesso numero di dati/record (altrimenti forma *n* sottogruppi con un numero di record quanto più simile possibile). Ad ogni sottogruppo viene associato un numero identificativo.

Viste materializzate

Le **viste materializzate** sono *viste* (interrogazioni) *il cui risultato viene pre-calcolato e memorizzato su disco*. Tali viste permettono di **velocizzare i tempi di risposta**, soprattutto per query che operano su una grande quantità di dati (E. tabella dei fatti di un DW): pre-calcolo degli aggregati (in particolare `group by`), join, etc.

L'accesso ad una vista materializzata è generalmente meno costoso, per tabelle di grosse dimensioni (Es. tabella dei fatti in join con una tabella dimensionale), rispetto all'esecuzione delle operazioni di join o al calcolo di aggregati.

Le viste materializzate sono solitamente associate a interrogazioni *frequenti* che operano *aggregazioni* su schemi di *grosse dimensioni*, ma possono essere usate anche per interrogazioni che non operano aggregazioni.

Viste materializzate e riscrittura delle interrogazioni

Una vista materializzata può essere usata in *qualunque interrogazione* di selezione come se fosse una **tabella**.

Come è noto, il DBMS può trasformare le interrogazioni al fine di ottimizzarne l'esecuzione. Nei sistemi DBMS attuali esiste tipicamente anche l'opzione di "**Queries rewriting**": le viste materializzate possono essere usate **automaticamente** in fase di riscrittura delle interrogazioni in modo **trasparente all'utente**.

Generalmente tali viste sono usate per risolvere interrogazioni simili a quella alla quale sono associate.

L'opzione di *Queries Rewriting* può essere abilitata in fase di definizione della vista.

Creazione viste materializzate

Per creare una vista materializzata, è necessario utilizzare la seguente istruzione SQL:

```
CREATE MATERIALIZED VIEW Nome
[BUILD {IMMEDIATE|DEFERRED}]
[REFRESH {COMPLETE|FAST|FORCE|NEVER}{ON COMMIT|ON DEMAND}]
[ENABLE QUERY REWRITE]
AS
    Query
```

La vista materializzata creata avrà come **schema** la lista degli attributi definiti nella clausola `SELECT` della *Query*.

Esempio di vista materializzata

Le tabelle di riferimento per l'esempio successivo presentano il seguente *schema*:

FRN(COD_F, Nome, Sede_F,)
ART(COD_A, Tipo, Colore, Peso)
PRG(COD_P, Nome, Sede_P)
FAP(COD_F, COD_P, COD_A, Q)

Si vuole "materializzare" la seguente interrogazione: *"Selezionare per ogni fornitore e per ogni articolo, la quantità totale fornita."*

```
SELECT Cod_F, Cod_A, SUM(Q)
FROM FAP
GROUP BY Cod_F, Cod_A;
```

Rispettando le seguenti opzioni:

- Caricamento dei dati *immediato*
- Refresh *completo* operato solo *su richiesta* dell'utente
- Abilitazione alla *riscrittura delle interrogazioni*

Soluzione

```
CREATE MATERIALIZED VIEW Frn_Art_sumQ
BUILD IMMEDIATE
REFRESH COMPLETE ON DEMAND
ENABLE QUERY REWRITE
AS
     SELECT Cod_F, Cod_A, SUM(Q)
     FROM FAP
     GROUP BY Cod_F, Cod_A;
```

Procedura per il refresh delle viste materializzate

L'utente, o un job di sistema, può richiedere il refresh del contenuto di una vista materializzata usando la procedura:

```
DBMS_MVIEW.REFRESH('vista', {'C'|'F'})
```

Dove:

- *vista*: nome vista da aggiornare.
- '*C*': refresh di tipo `COMPLETE`.
- '*F*': refresh di tipo `FAST`.

E' necessario tener traccia del ROWID perché se vi è stato un aggiornamento di una tabella è necessario sapere qual è il record che ha subito la variazione; è inoltre necessario conoscere la sequenza temporale con cui le operazioni si sono susseguite.

In sintesi:

- *SEQUENCE*: contatore che indica la sequenza con la quale si sono susseguite le operazioni
- *ROWID*: indica quale record che è stato modificato

Il materialized view log può essere quindi definito come segue:

```
CREATE MATERIALIZED VIEW LOG ON FAP
WITH SEQUENCE, ROWID (Cod_F, Cod_A, Q)
INCLUDING NEW VALUES;
```

Esempio di vista materializzata con opzione fast refresh

Si vuole materializzare l'interrogazione:

```
SELECT Cod_F, Cod_A, SUM(Q)
FROM FAP
GROUP BY Cod_F, Cod_A;
```

Opzioni

- Caricamento dei dati *immediato*
- *Fast refresh* eseguita *automaticamente* dopo ogni commit
- Abilitazione alla *riscrittura delle interrogazioni*

Soluzione

Creazione del materialized view log associato alla tabella *Fast*:

```
CREATE MATERIALIZED VIEW LOG ON FAP
WITH SEQUENCE, ROWID (Cod_F, Cod_A, Q)
INCLUDING NEW VALUES;
```

Creazione della vista materializzata:

```
CREATE MATERIALIZED VIEW Frn_Art_sumQ
BUILD IMMEDIATE
REFRESH FAST ON COOMIT
ENABLE QUERY REWRITE
AS
     SELECT Cod_F, Cod_A, SUM(Q)
     FROM FAP
     GROUP BY Cod_F, Cod_A;
```

Defining the Window size

The window (or the set of rows to be worked on) can be defined as a fixed number of rows, all preceeding or all following rows or it can be calculated based on comparing values (or time periods) in the current row with values in the ordered sequence. This definition is made with the `ROWS` or `RANGE` clause

Syntax:

```
Function([arguments]) OVER
   ([PARTITION BY value/expr]
       [ORDER BY expr [ASC|DESC]
          [ROWS | RANGE windowing_clause]])

windowing_clauses:

   INTERVAL 'nn' DAY PRECEDING
   INTERVAL 'nn' SECONDS FOLLOWING
   INTERVAL 'nn' MONTH PRECEDING
   BETWEEN x PRECEDING AND y FOLLOWING
   BETWEEN x PRECEDING AND y PRECEDING
   BETWEEN CURRENT ROW AND y FOLLOWING
   BETWEEN x PRECEDING AND CURRENT ROW
   BETWEEN x PRECEDING AND UNBOUNDED FOLLOWING
   BETWEEN UNBOUNDED PRECEDING AND y FOLLOWING
   column BETWEEN current.column +/- n AND current.column +/- m
   UNBOUNDED PRECEDING | FOLLOWING
   value/expr PRECEDING | FOLLOWING
   CURRENT ROW
```

For time intervals, the `ORDER BY` clause has to be a `DATE` column/expression.

If you omit the windowing_clause entirely, the default is:

```
RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW.
```

- If `ROWS` is specified, it's a physical offset (the no. of rows in the window)
- If `RANGE` is specified, it's a logical offset

`UNBOUNDED` means the very first/last row of the partition, or if not partitioned, the first/last row of the dataset.

Introduzione al corso

I sistemi classici di gestione di basi di dati vanno normalmente sotto il nome di una grande famiglia detta OLTP.

OLTP - On Line Transaction Processing

Gli OLTP sono sistemi che permettono di gestire una serie di operazioni volte all'erogazione di servizi; quindi la base di dati offre supporto per memorizzare i dati che vengono utilizzati poi durante il processo di gestione. I dati sono quindi messi a disposizione di un processo aziendale o pubblico.

Ogni operazione svolta sui dati è una **transazione**.

Come sono fatti i dati?

Questi sistemi hanno necessità di memorizzare delle "fotografie" del **valore corrente** dei dati presenti nella base di dati. I dati sono generalmente **molto dettagliati** e sono organizzati in base al **modello relazionale** (con centinaia di tabelle diverse).

Come sono fatte le operazioni?

Le operazioni sono **strutturate** e **ripetitive** (Es. prenotazione ad un esame Polito); quando queste operazioni vengono definite sono strutturate in un ordine ben preciso e sono ripetute molte volte sempre nello stesso modo, cambiando solo alcuni parametri.

Avendo sempre la stessa operazioni che viene ripetuta tante volte, è possibile ottimizzare ed accelerare tutte le operazioni in modo stabile da un punto di vista della progettazione fisica.

Le operazioni di tipo gestionale in questo tipo di sistemi sono tipicamente azioni di lettura e scrittura che accedono a pochi record (nell'ordine della decina); i programmi sono quindi costituiti da **brevi transazioni** sui dati che fanno degli accessi rapidi ad essi.

Gli OLTP, così come tutti i sistemi transazionali classici, devono garantire il rispetto delle proprietà **ACID**: questo è un aspetto critico. Esse sono tipicamente garantite da blocchi opportuni del DBMS.

I volumi di dati trattati da questo tipo di sistemi sono corposi, ma non esageratamente grandi, nell'ordine delle centinaia di MB o decine di GB (dimensione dei database ~ **100MB – 10GB**).

Vediamo ora delle basi di dati che devono essere in grado di supportare delle operazioni di tipo analitico, ovvero di analisi dei dati. Questo tipo di operazioni richiedono una tipologia di accesso ai dati completamente diverso da quelle classiche.

© Proprietà riservata dell'autore - Digitalizzazione e distribuzione a cura del CENTRO APPUNTI - Corso Luigi Einaudi, 55 - Torino / Pagina 57 di 598

Introduzione al DBMS

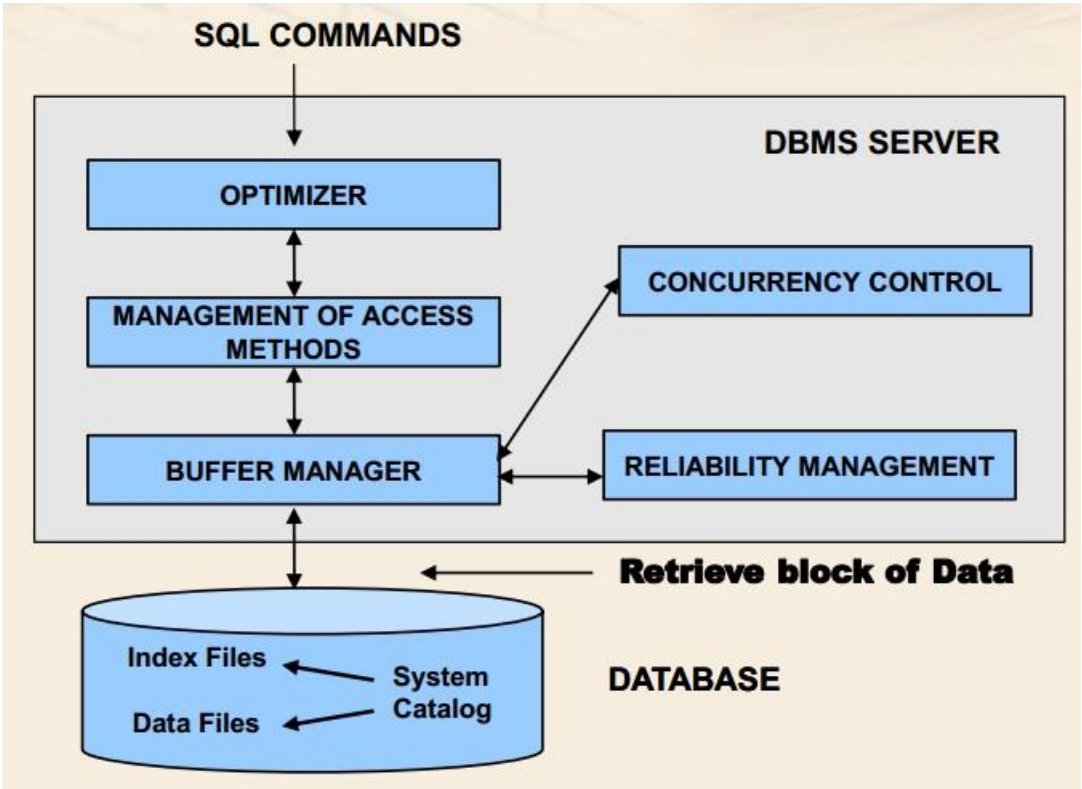
Il **DBMS** (Data Base Management System) è un pacchetto software molto complesso progettato per memorizzare e gestire i database.

Noi saremo interessati ai meccanismi interni di un DBMS tramite i quali vengono forniti i servizi alle varie applicazioni. Conoscere questi meccanismi permette di effettuare le scelte di progettazione più corrette, quali: configurazione del sistema e progettazione fisica delle applicazioni.

Alcuni dei servizi messi a disposizione dei DBMS stanno diventando disponibili anche nei comuni sistemi operativi.

Architettura dei DBMS

In figura è mostrata un'architettura per grandi linee di un DBMS, che è generalmente costituito da più **blocchi**:



Ogni blocco che fa parte del DBMS offre dei servizi specifici.

La base di dati (**database**) è costituita dalla collezione di uno o più file memorizzati con una struttura proprietaria che può essere differente da DB a DB.

L'**optimizer** è il primo blocco che riceve l'istruzione SQL e che permette di garantire una delle proprietà fondamentali dei sistemi relazionali: l'**indipendenza dei dati**.

Buffer Manager - Concurrency Control - Reliability Manager

Il **buffer manager** gestisce il trasferimento di dati (pagine) da disco a memoria centrale e viceversa. Esso riceve delle richieste di pagine dai moduli superiori e si occupa di effettuarne il trasferimento senza però conoscerne la struttura interna, cioè senza conoscere i dati in esse contenuti.

Tale modulo si interfaccia quindi al S.O. per richiedere le letture da disco e per gestire l'allocazione opportuna della memoria. Il S.O. definisce infatti una porzione di memoria centrale che viene pre-allocata per il DBMS quando entra in esecuzione e che viene direttamente gestita dal buffer manager e non più dall'S.O (Es. Oracle ha una grossa area di memoria chiamata SGA).

I blocchi di memoria pre-allocati per il DBMS sono **condivisi** da diverse applicazioni ed è proprio il buffer manager (insieme al blocco di concurrency control) a consentirne la condivisione. Il buffer manager infatti interagisce con:

- Il gestore dell'accesso concorrente (**concurrency control**) – blocco che si occupa di gestire gli accessi concorrenti ai dati, situazione particolarmente critica per le operazioni di scrittura. Esso garantisce inoltre che diverse applicazioni non interferiscano tra loro, evitando i problemi di consistenza dei dati.

  Questo blocco consente quindi di ottenere la **proprietà di isolamento** delle transazioni grazie alla sincronizzazione delle operazioni di accesso ai dati.

- Il gestore dell'affidabilità (**reliability manager**) – blocco che si occupa di garantire la correttezza e persistenza dei dati contenuti nel database quando il sistema è soggetto a guasti; è infatti in grado di recuperare dati che risultano apparentemente persi per via di un guasto. Per far ciò utilizza delle strutture ausiliarie (**file di log**) che permettono di ripristinare lo stato del sistema al momento del guasto.

  Esso quindi garantisce la **persistenza** dei dati e l'esecuzione **atomica** delle transazioni (anche in caso di guasti).

Delimitazione delle transazioni

L'*inizio* di una transazione è tipicamente implicito e coincide con la prima istruzione SQL all'inizio di un programma o la prima istruzione SQL subito dopo la fine della transazione precedente.

La *fine* di una transazione può essere invece identificata da due comandi:

- **`COMMIT (WORK)`** – la transazione è stata completata con successo (senza errori) e le modifiche sono state apportate alla base di dati.
- **`ROLLBACK`** – la transazione è stata annullata per via di un errore. Tutte le operazioni già eseguite dalla transazione vengono annullate riportando la base di dati allo stato precedente all'inizio della transazione stessa.

Generalmente il 99.9% delle transazioni si concludono con un **commit work**. Le restanti transazioni vengono terminate con un **rollback** che può essere distinto in:

- Rollback richiesto dalla transazione ("suicidio" o "suicide") – la transazione termina per via di un errore interno al programma stesso.
- Rollback richiesto dal sistema ("omicidio" o "murder") – la transazione è terminata dal sistema in maniera forzata.

Proprietà delle transazioni

Le proprietà **ACID** delle transazioni sono le seguenti:

- **A**tomicità;
- **C**onsistenza;
- **I**solamento;
- **D**urabilità (o persistenza).

Atomicità

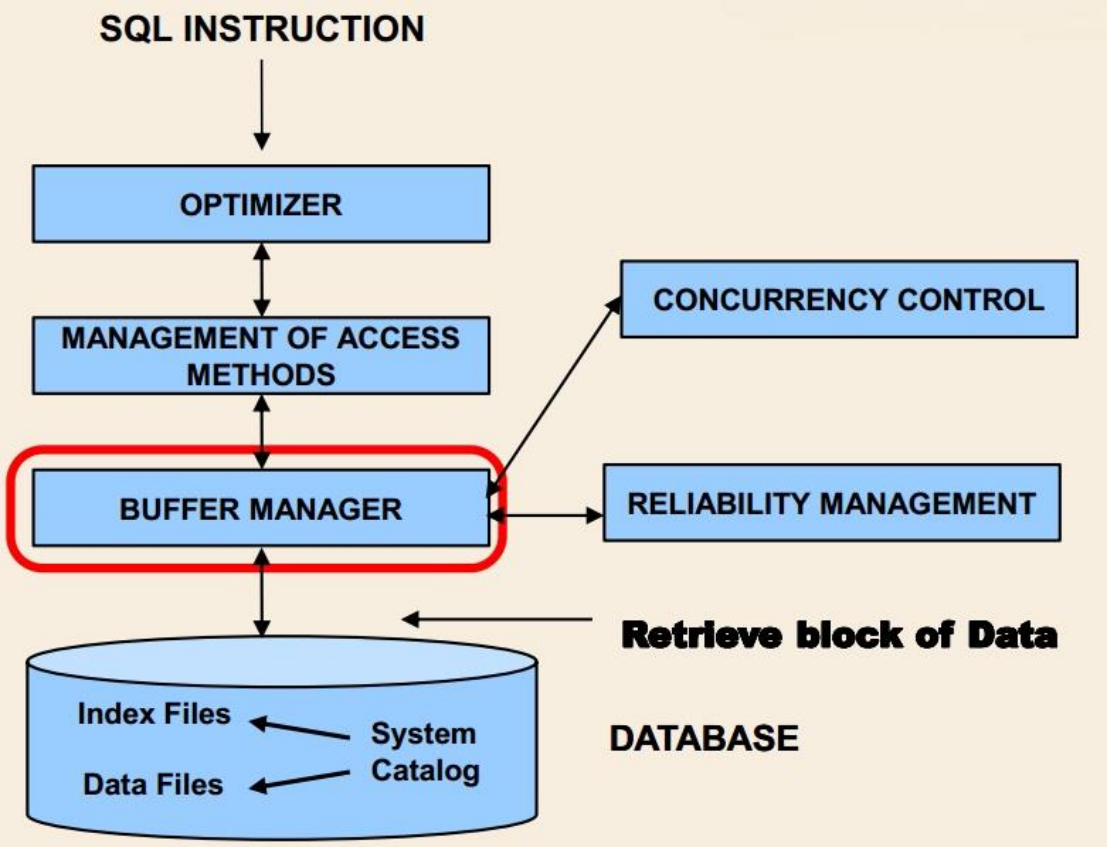
Una transazione non può essere divisa in unità più piccole, ovvero in sotto-blocchi di istruzioni separate. Non è quindi possibile lasciare il database in uno stato intermedio causato dall'esecuzione di una parte della transazione.

Questa proprietà è garantita dalle primitive **UNDO** e **REDO**.

- `UNDO` – Il sistema annulla tutte le operazioni eseguite da una transazione fino al punto corrente (in cui è tipicamente avvenuto l'errore). Questa operazione è utilizzata in caso di rollback.

© Proprietà riservata dell'autore - Digitalizzazione e distribuzione a cura del CENTRO APPUNTI - Corso Luigi Einaudi, 55 - Torino / Pagina 63 di 598

Buffer manager



Il **buffer manager** è responsabile della gestione del buffer di memoria assegnato al DBMS; in pratica gestisce il trasferimento di pagine dal disco alla memoria centrale e viceversa. Esso non conosce il contenuto delle pagine, si occupa solo della loro gestione.

Una gestione efficiente del buffer è fondamentale per avere una elevata efficienza dell'intero DBMS.

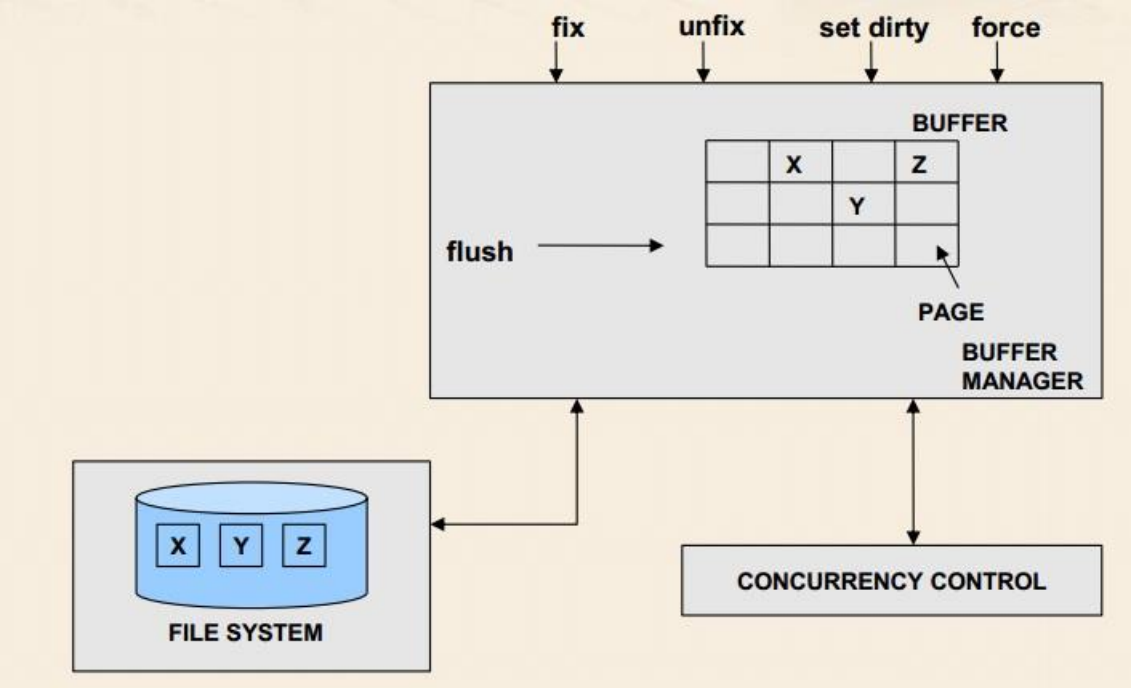
DBMS Buffer

Il **buffer** del DBMS è un grande blocco di memoria centrale gestito dal buffer manager che viene pre-allocato dal sistema operativo per il DBMS stesso ed è condiviso tra le diverse transazioni in esecuzione.
Il buffer manager permette poi alle singole transazioni di accedere alle pagine presenti nel buffer tramite delle opportune primitive.

La memoria all'interno del buffer è organizzata in *pagine*, la cui dimensioni dipendono dalla dimensione dei blocchi di I/O del S.O. e dal DBMS.

Architettura del Buffer Manager

Nella figura successiva è riportata una descrizione sommaria dell'architettura del buffer manager:



Appare quindi evidente il ruolo del Buffer Manager che ha il compito di caricare in memoria le pagine dal disco (File System) e restituire al blocco superiore (Es. Gestore dell'accesso, Ottimizzatore, Transazione) un puntatore ad esse.

Il buffer manager si interfaccia inoltre al *gestore dell'accesso concorrente* al fine di verificare l'autorizzazione di ogni transazione ad accedere alla pagina richiesta: verifica se la transazione che richiede di accedere ad una determinata pagina di memoria è autorizzata a farlo oppure no, perché magari la pagina è già utilizzata da un'altra transazione.

Le primitive

Il buffer manager mette a disposizioni degli altri blocchi una serie di *primitive* che permettono di accedere ai diversi metodi che possono essere utilizzati per caricare le pagine da disco a memoria centrale e viceversa:

- Fix;
- Unfix;
- Force;
- Set Dirty;
- Flush.

Queste primitive richiedono l'autorizzazione di accesso condiviso da parte del concurrency control manager.

Infine, la pagina richiesta viene *caricata nel buffer* di memoria sovrascrivendo la pagina scelta per essere sostituita, e il suo *indirizzo viene restituito* alla transazione che ha invocato la primitiva (transazione richiedente).

Se non sono presenti *né pagine libere, né pagine non libere con Count=0* vuol dire che il sistema sta andando verso una congestione e la transazione richiedente deve essere messa *in attesa* fino a che non vi è la possibilità di caricare in memoria la pagina richiesta.

***Le **pagine non libere** sono pagine assegnate a transazioni ancora in esecuzione.*

****Le **pagine non libere con Count=0** sono pagine assegnate a transazioni ancora in esecuzione (non terminate) ma che hanno già rilasciato la pagina, ovvero a transazioni non attive rispetto a tale pagina.*

Primitiva Unfix

La primitiva *Unfix* viene utilizzata dalle transazioni per comunicare al buffer manager che una determinata pagina non è più utilizzata dalla transazione stessa.

A seguito dell'esecuzione di tale primitiva, la variabile di stato *Count* associata alla pagina in questione sarà decrementata di 1.

Primitiva Set Dirty

La primitiva *Set Dirty* viene utilizzata dalle transazioni per comunicare al buffer manager che una determinata pagina è stata modifica dalla transazione stessa: la variabile di stato *Dirty* viene settata ad 1 e sarà necessario copiare la pagina su disco prima di sovrascriverla.

Primitiva Force

La primitiva *Force* viene utilizzata dalle transazioni per richiedere un trasferimento **sincrono** di una o più pagine su disco; la transazione richiedente viene quindi sospesa fino al termine della primitiva, ovvero resta in attesa che il buffer manager completi l'operazione di scrittura su disco.

Questa primitiva comporta **sempre una scrittura su disco**.

Tale primitiva risulta particolarmente costosa in quanto comporta una *scrittura sincrona* su disco che obbliga la transazione richiedente a restare in attesa del completamento dell'operazione. Se ogni transazione rimanesse sempre in attesa della scrittura su disco, l'efficienza globale del sistema si ridurrebbe notevolmente.

Force/No Force

- **Force** – Tutte le pagine attive di una transazione sono scritte in maniera **sincrona** su disco dal buffer manager durante l'operazione di commit della transazione.

  Questo comporta l'esecuzione di più scritture su disco: ogni record modificato di una pagina porterà alla copia dell'intera pagina su disco (al commi della transazione), indipendentemente dal fatto che un altro record della pagina potrebbe essere modificato di lì a poco da un'altra transazione → costo maggiore: transazioni eseguite più lentamente [☹]

- **No Force** – Le pagine sono scritte in maniera **asincrona** su disco dal buffer manager tramite la primitiva Flush oppure quando è necessario liberare il buffer per servire una richiesta di Fix.

  Le pagine che dipendono da transazioni concluse con successo (andate in commit work) potrebbero quindi essere scritte su disco **dopo** che la transazione è terminata. Risulta quindi evidente che in caso di *guasti di sistema*, i cambiamenti apportati dalla transazione dovranno essere ri-eseguiti (operazione di REDO) per renderli definitivi su disco: è necessario che il sistema sia in grado di ri-eseguire tutte le attività che potrebbero essere andate perse in quanto non ancora salvate su disco → Necessaria primitiva di **REDO**.

Nei sistemi commerciale le politiche più utilizzate sono tipicamente di tipo *steal/no force* per via della loro efficienza (**throughput maggiore** di transazioni):

- La politica **no force** garantisce migliori prestazioni in termini di operazioni I/O;
- La politica **steal** potrebbe essere necessaria nel caso di query che accedono ad un grande numero di pagine.

Accesso fisico ai dati

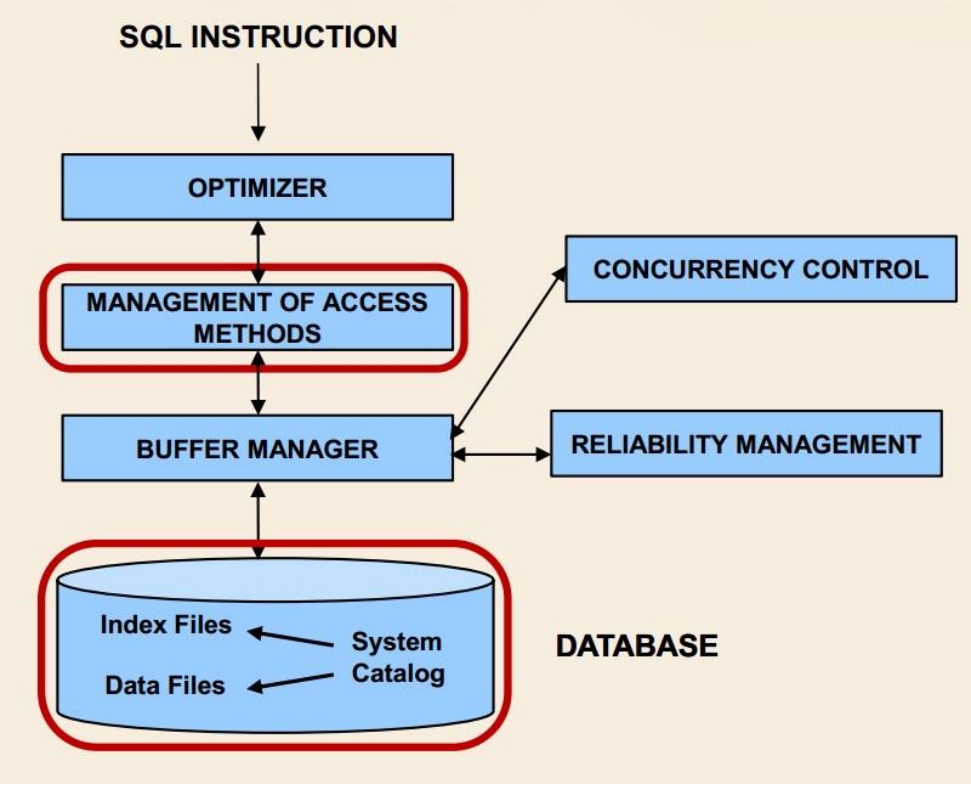
Strutture di accesso fisico

I dati possono essere memorizzai su disco in formati differenti al fine di garantire una esecuzione efficiente delle query; formati differenti sono adatti a soddisfare i diversi bisogni delle singole query.

Le **strutture di accesso fisico** ai dati descrivono come questi sono memorizzati su disco.

Gestore dei metodi di accesso

Il gestore dei metodi di accesso è il blocco del DBMS che conosce il modo in cui i dati sono stati fisicamente scritti su disco. Esso è predisposto a conoscere e gestire la struttura dei dati salvati nel database e degli eventuali indici esterni ad essi associati.



Il **gestore dei metodi di accesso** trasforma *un piano di accesso*, generato dall'optimizer, in una *sequenza di accessi fisici* necessari per accedere alle pagine su disco (database). Esso sfrutta diversi *metodi di accesso* a seconda della struttura fisica da utilizzare.

Metodo di accesso

Un **metodo di accesso** è un modulo software specifico per una singola struttura fisica di dati. Esso fornisce le primitive necessarie a *leggere* e *scrivere* i dati.

Strutture fisiche di accesso

Le **strutture fisiche di accesso** descrivono come i dati sono memorizzati su disco al fine di garantire una esecuzione efficiente delle query (in particolare istruzioni DML: `SELECT` SQL, `UPDATE` SQL, etc.).

Nei sistemi relazionali ci sono diversi aspetti che concorrono al fine di rendere efficiente l'accesso ai dati:

- La struttura fisica di memorizzazione dei dati (ovvero delle tabelle):
  - Strutture sequenziali;
  - Strutture hash.

- La modalità di gestione degli indici*:
  - Strutture ad albero (B-Tree, B^+-Tree);
  - Unclustered hash index;
  - Indici Bitmap.

*Un **indice** è una struttura fisica accessoria che consente di reperire più velocemente le informazioni presenti in un altro file (Es. tabella).

Come possono essere memorizzate le tabelle?

Struttura sequenziale

In una *struttura sequenziale* le tuple sono memorizzate in base ad un determinato ordine sequenziale.
Diversi tipi di strutture implementano diversi criteri di ordinamento.

Le strutture sequenziali disponibili sono:

- Heap File (entry sequenced);
- Ordered sequential structure;

Heap file
In una struttura *heap file* le tuple sono memorizzate nel file in modo sequenziale in base all'ordine di inserimento: l'operazione di inserimento è tipicamente una operazione di `APPEND` alla fine del file.

Vantaggi
Tutto lo spazio di ogni blocco è riempito completamente prima di passare al blocco successivo [☺], ne consegue che le operazioni di lettura e scrittura sequenziali risultano particolarmente efficienti [☺].

© Proprietà riservata dell'autore - Digitalizzazione e distribuzione a cura del CENTRO APPUNTI - Corso Luigi Einaudi, 55 - Torino / Pagina 75 di 598

In questo modo si perde però l'ordinamento corretto dei dati rispetto alla sort key e il tempo di accesso alle tuple aumenta in quanto è necessario accedere sia al blocco ordinato che al file di overflow.

Le strutture sequenziali ordinate sono tipicamente utilizzate con indici (primari) organizzati secondo una struttura di tipo B^+-Tree clustered la cui chiave risulta essere la sorter key.

Questa soluzione è generalmente adottata dai DBMS per memorizzare su disco i risultati intermedi delle operazioni.

Strutture ad albero

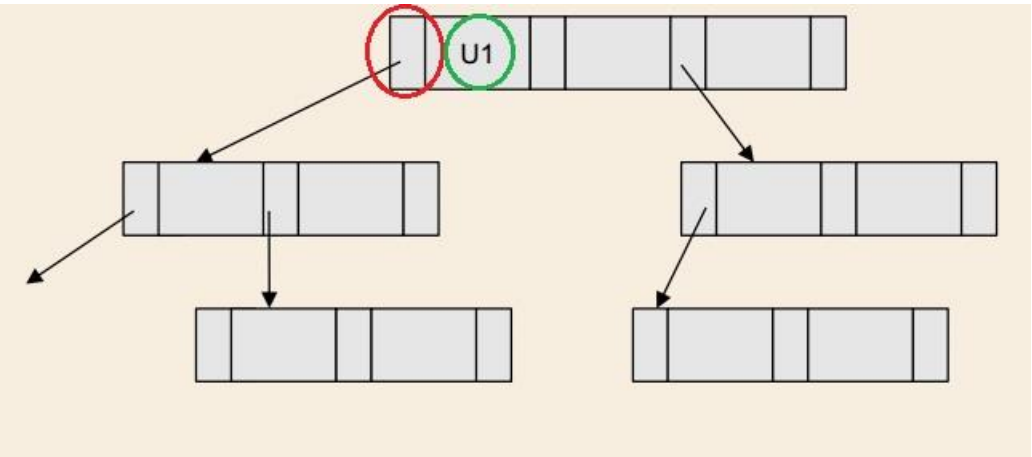
Le strutture ad albero forniscono un accesso "diretto" ai dati [☺] basato sul valore di un campo chiave che può includere uno o più attributi.

Questo tipo di strutture non vincolano la posizione fisica delle tuple [☺] ed è per questo che risultano essere le strutture più largamente diffuse nei DBMS relazionali.

Le strutture più usate nei DBMS sono le strutture ad albero e quelle heap file. Le strutture ad albero vengono utilizzate sia per memorizzare i dati delle tabelle secondo un particolare ordine creando un file sequenziale, sia per memorizzare gli indici ad esse associati, ovvero creare una struttura indiretta (separata dai dati) che permette di reperire velocemente i dati stessi.

Caratteristiche generali
La struttura ad albero è costituita da *un nodo radice* e *tanti nodi intermedi*. I nodi hanno tipicamente un **fan-out** elevato, ovvero ogni nodo possiede molti "nodi figli".

I *nodi foglia* dell'albero sono quelli che consentono l'accesso ai dati.



Le porzioni del nodo analoghe a quella marcata in rosso sono dei **puntatori fisici** ai nodi successivi, mentre il **valore** rispetto al quale cercare il dato è contenuto nella porzione di nodo evidenziata in verde.

© Proprietà riservata dell'autore - Digitalizzazione e distribuzione a cura del CENTRO APPUNTI - Corso Luigi Einaudi, 55 - Torino / Pagina 77 di 598

In pratica gli alberi B^+-tree hanno dei puntatori (in figura evidenziati in rosso) che collegano in modo sequenziale tutti i nodi foglia, permettendo una navigazione dell'albero che non deve necessariamente partire dal nodo radice e scendere ai nodi più bassi, ma può essere eseguita scorrendo le foglie una dopo l'altra.

Si parla di **B**-Tree e **B**$^+$-tree in quanto si parla di alberi bilanciati (B = **Balanced**). Questo significa che:

- Tutte le foglie sono alla *stessa distanza* dal nodo radice;
- Il *tempo di accesso* ai dati (tempo di attraversamento dell'albero) è costante, indipendentemente dal valore cercato.

Possono essere necessarie delle operazioni di modifica dell'albero in caso di inserimenti o cancellazioni, al fine di mantenere effettivamente bilanciato l'albero. Queste operazioni di gestione sono particolarmente costose.

Per far si che le operazioni di "ri-bilanciamento" non siano necessarie di frequente, tipicamente i vari nodi presentano un fan-out elevato, in modo tale che se i nodi non vengono riempiti completamente in fase di creazione dell'albero, nelle fasi successive sarà possibile riempirli ulteriormente senza necessità di effettuare il bilanciamento.

Per cui, sarà necessario ri-bilanciare l'albero nelle seguenti situazioni:

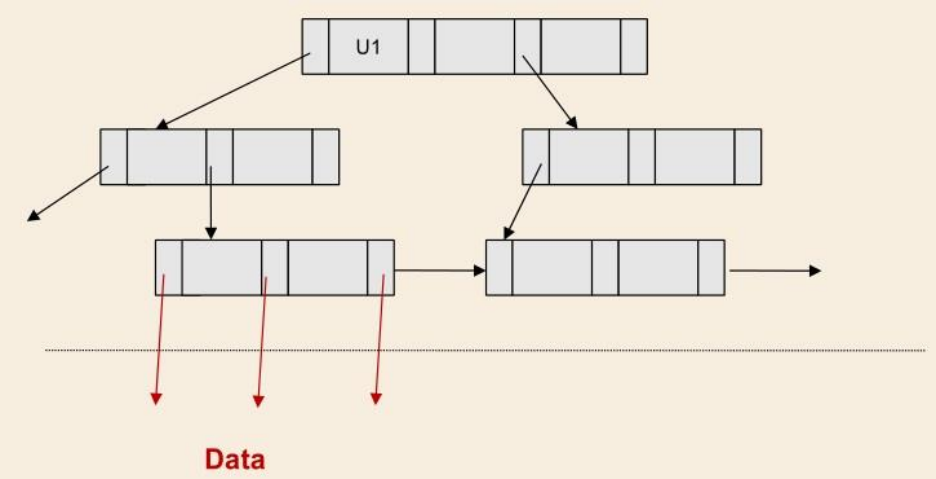
- In **inserimento**: quando i nodi intermedi diventano pieni sarà necessario separarli (splitting) e ri-bilanciare l'albero;
- In **cancellazione**: quando i nodi diventano troppo vuoti sarà necessario accorparli e ri-bilanciare l'albero.

Clustered Trees

Negli alberi di tipo *clustered* (anche detti a **memorizzazione primaria**) le tuple sono contenute all'interno dei nodi foglia; si può quindi scegliere un unico insieme di attributi rispetto ai quali effettuare l'ordinamento, cioè non è possibile definire più di un albero clustered su una tabella. Questo perché la posizione fisica di ogni tupla è vincolata al nodo foglia in cui è memorizzata. Tale posizione può essere modificata effettuando lo *splitting* (divisione) del nodo quando questo è pieno.

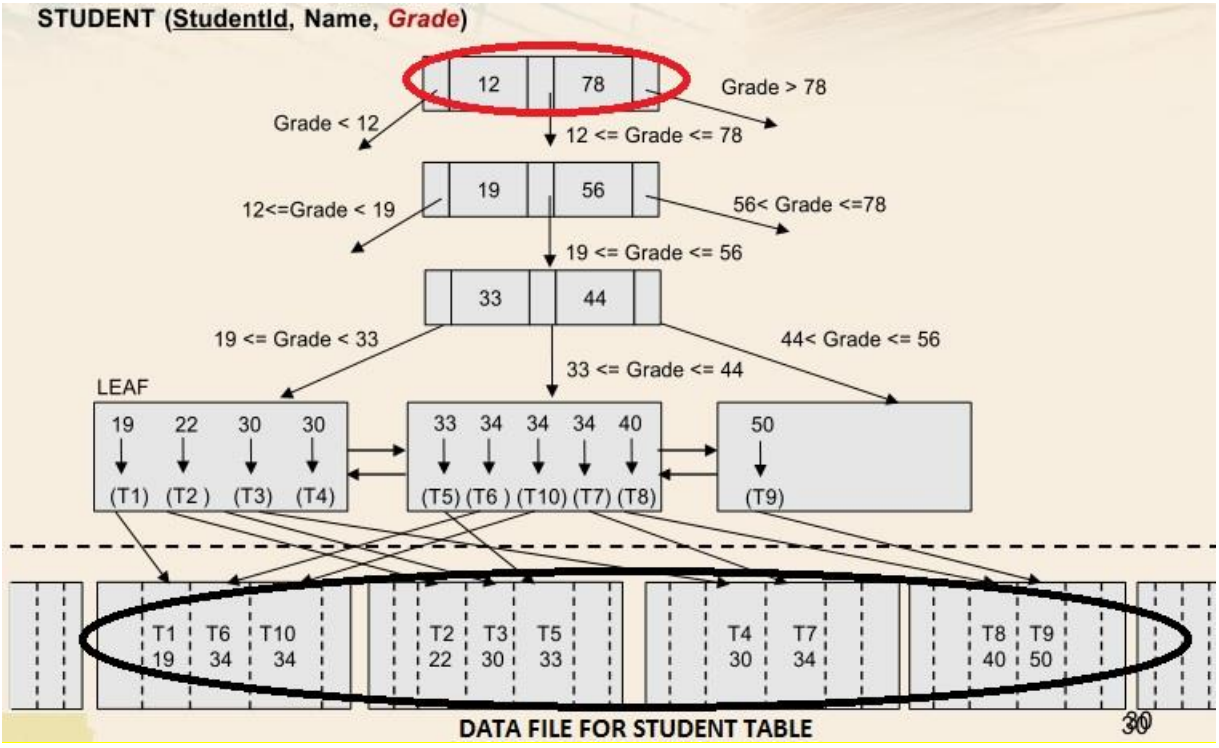
Un albero di questo tipo è anche chiamato "**key sequenced**" (essendo organizzato in sequenza sulla chiave) ed è tipicamente utilizzato per imporre un ordinamento fisico sui dati che sia *diverso dall'ordine di arrivo*.
Normalmente è possibile utilizzare questa struttura dati per indicizzare la chiave primaria.

Un esempio di B^+-Tree unclustered è il seguente:



Esempio: Indice B^+-Tree unclustered

Nella figura successiva è mostrato un esempio di indice B^+-Tree unclustered nel quale il nodo foglia contiene il valore dell'attributo rispetto al quale è eseguita l'indicizzazione e il puntatore fisico al dato.



Si suppone che il voto (grade) vari tra 0 e 100.

In basso (evidenziato in **nero**) è presente il file contenente le tuple ordinate secondo un ordine generico. In alto (evidenziato in rosso) vi è il nodo radice che esegue una prima suddivisione delle tuple in funzione di intervalli di voto. Ognuno dei nodi intermedi andrà poi a curare con intervalli più piccoli e precisi l'intervallo di partenza identificato dal puntatore che dirige verso il nodo stesso.

Indici B/B$^+$-Tree: Vantaggi e Svantaggi

Vantaggi [☺]:

- Molto efficienti per le query su intervalli di valori;
- Adatti per scansioni sequenziali eseguite in ordine rispetto al campo chiave: sempre valido per gli alberi clustered, non garantito negli altri casi.

Svantaggi [☹]:

- Gli inserimenti possono richiedere lo split di una foglia o di nodi intermedi; questo tipo di operazione comporta un costo computazionale elevato.
- Le cancellazioni possono richiedere la fusione di nodi poco "popolati" e il ribilanciamento dell'albero.

Strutture Hash

Le strutture hash garantiscono un accesso diretto ed efficiente ai dati rispetto al valore di un **campo chiave**; tale campo chiave può includere uno o più attributi.

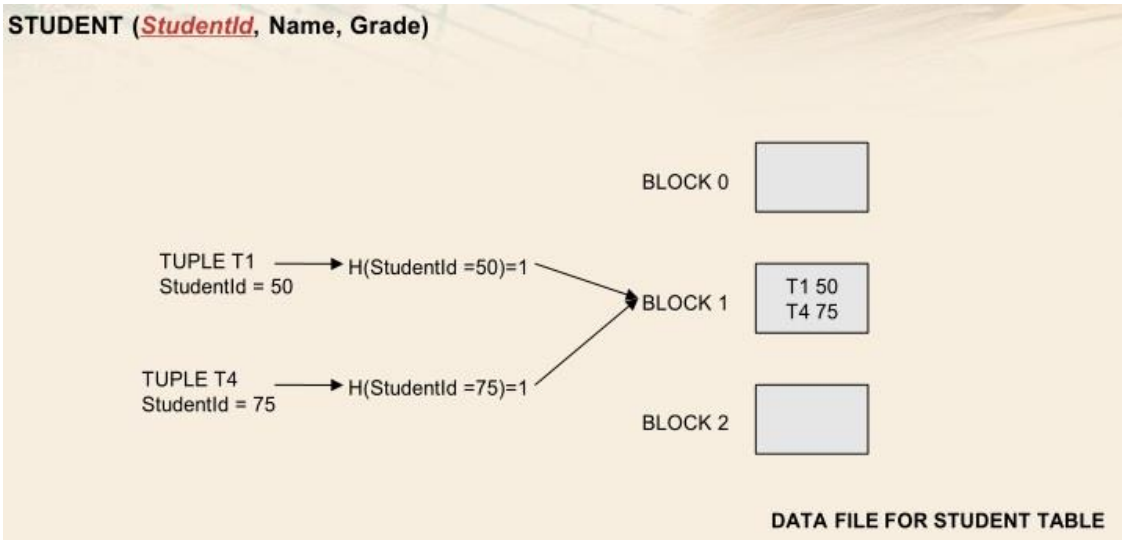
In pratica viene applicata una funzione di hash ai valori da memorizzare e il risultato ottenuto viene utilizzato per il posizionamento dei dati stessi.

Supponiamo che la struttura hash abbiamo *B* blocchi. La funzione hash viene applicata al campo chiave di un record e restituirà un valore compreso tra *0* e *B-1*; tale valore identifica la posizione del record nella struttura dati (cioè il numero di un blocco).

I blocchi non dovrebbero mai essere completamente pieni al fine di garantire la dinamicità della struttura dati, ovvero la possibilità di inserimento di nuovi dati in maniera efficiente.

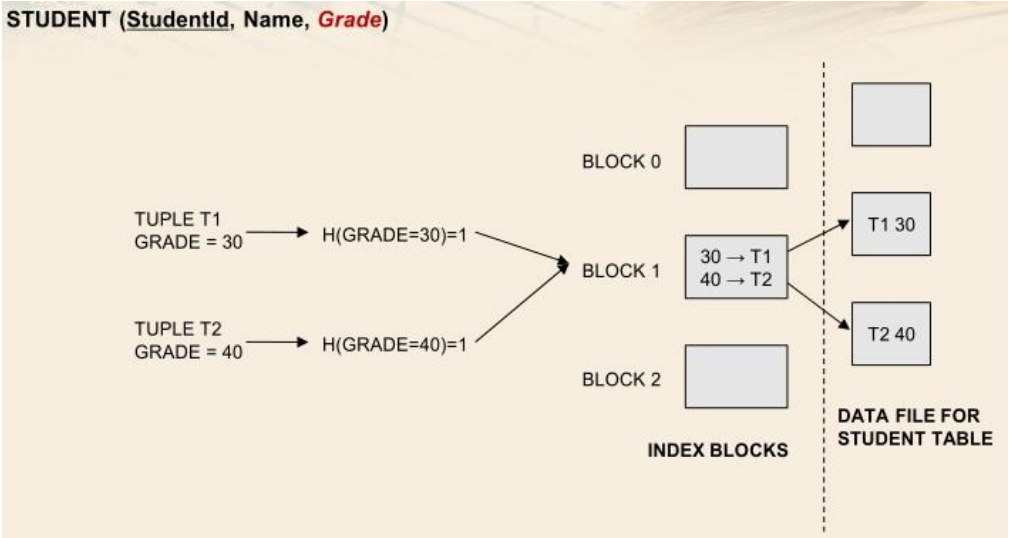
Esempio: Indice Hash

Nella figura successiva è mostrato un esempio di indice Hash.



Esempio: Indice Hash Unclustered

Nell'esempio seguente è presene un file dati che contiene la tabella, al quale si va a sovrapporre l'indice costituito dai suoi blocchi. Quando si va a cercare o inserire una tupla con un determinato valore, sarà necessario applicare la funzione di hash che permetterà di accedere al blocco contenente il puntatore alla tupla stessa.



Si osservi che in questo caso l'indice è costruito sul voto e non sulla matricola, in quanto un indice secondario ha più senso costruito su tale attributo rispetto ad un indice primario invece costruito sulla chiave primaria.

Indice Hash: Vantaggi e Svantaggi

Vantaggi [☺]:

- Molto efficienti per query con predicati semplici di uguaglianza (del tipo ***attributo=valore***) rispetto alla chiave. Con un solo accesso al blocco è infatti possibile accedere a tutti i record (o ai puntatori relativi ai record) che hanno quel determinato valore.
- Non è richiesto **l'ordinamento** dei blocchi del disco.

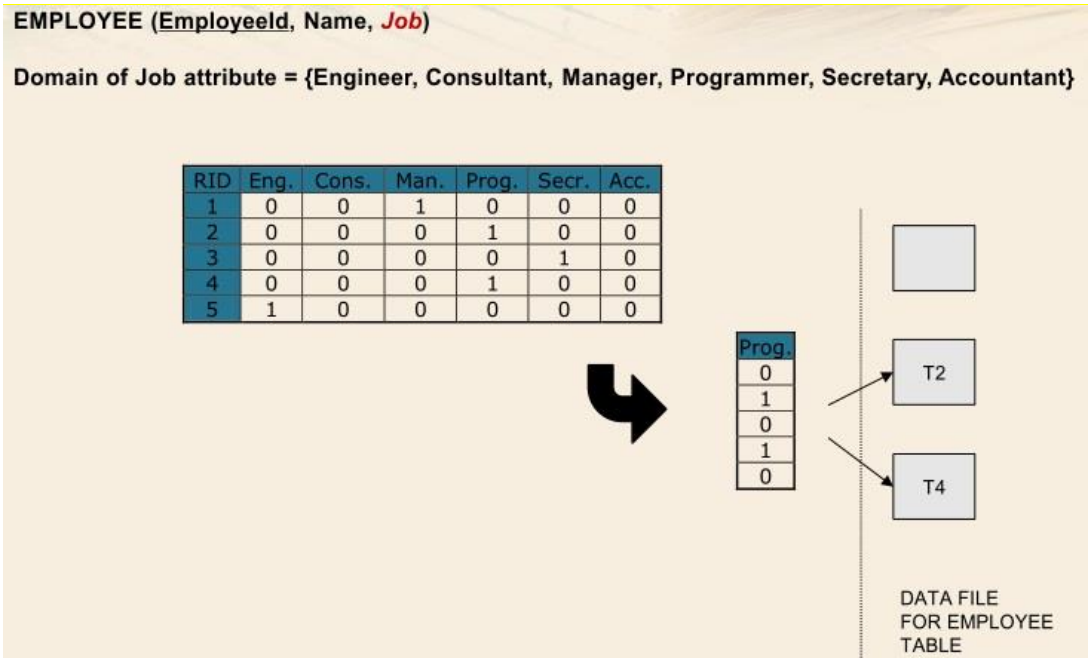
Svantaggi [☹]:

- Inefficienti per query ad **intervalli**; non ha quindi senso applicare un indice hash ad un attributo sul quale si fanno tipicamente ricerche ad intervalli.
- Si potrebbero verificare fenomeni di **collisione** nel caso in cui i blocchi siano pieni. Tale situazione è ovviamente più critica se l'indice è clustered rispetto al caso unclustered.

Esempio: Indice Bitmap

Nella figura successiva è mostrato un esempio di applicazione dell'indice bitmap. Si ha una tabella di impiegati caratterizzati da una serie di attributi ognuno con il proprio domino.

Si supponga di cercare tutte le tuple di impiegati con una determinata mansione; in tal caso è sufficiente estrarre la colonna dell'indice relativa a quella determinata mansione e selezionare le sole tuple che hanno il bit pari ad 1 in corrispondenza di essa.



Si osservi che all'interno della matrice di bit non sono contenuti dei puntatori ai dati fisici in quanto ci sarebbe un eccessivo impiego di spazio; questo significa che il sistema, a partire dalla posizione (i, j) e dall'RID ad essa relativo, deve essere in grado di accedere su disco alla tupla corrispondente.

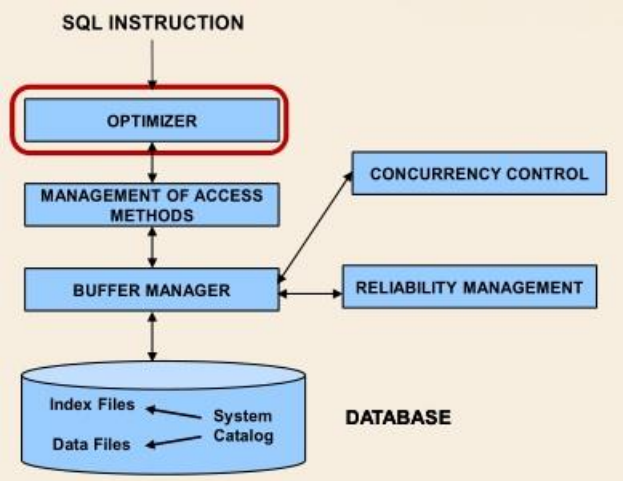
Indice Bitmap: Vantaggi e Svantaggi

Vantaggi [☺]:

- Molto efficienti per interrogazioni con predicati semplici del tipo ***attributo=valore***, dove l'attributo presenta una cardinalità bassa di dominio.
- Molto efficienti per espressioni booleane di predicati, è infatti possibile ridurre tali espressioni in operazioni tra bit. In pratica è possibile eseguire AND/OR bit a bit tra le varie colonne al fine di risolvere in maniera più efficiente la condizione e quindi velocizzare la ricerca; si va infatti a ridurre il numero di blocchi che è necessario leggere, rispetto alla situazione che si avrebbe eseguendo AND/OR dopo aver effettuato la ricerca sui dati.
- Adatto ad attributi aventi un dominio con cardinalità limitata.

Ottimizzazione delle Query

Il modulo del DBMS che si occupa di eseguire l'ottimizzazione delle query, avvalendosi delle strutture di ottimizzazione fin qui analizzate, è l'**optimizer**. Esso si occupa di scegliere una strategia (Es. Quale strutture dati utilizzare) efficiente per l'esecuzione delle query ed è per questo uno dei moduli fondamentali di un DBMS relazionale.



Come già accennato in precedenza l'optimizer garantisce la proprietà di **indipendenza dei dati,** ovvero:

- La forma con la quale sono scritte le query SQL non vincola il modo in cui queste vengono poi implementate;
- Una riorganizzazione fisica dei dati (Es. cambiare le performance, modificare l'architettura, distribuire i dati su nodi diversi) non richiede la riscrittura delle query SQL e in genere di tutte le applicazioni definite su di essi (purché non venga modificata la struttura logica delle tabelle).

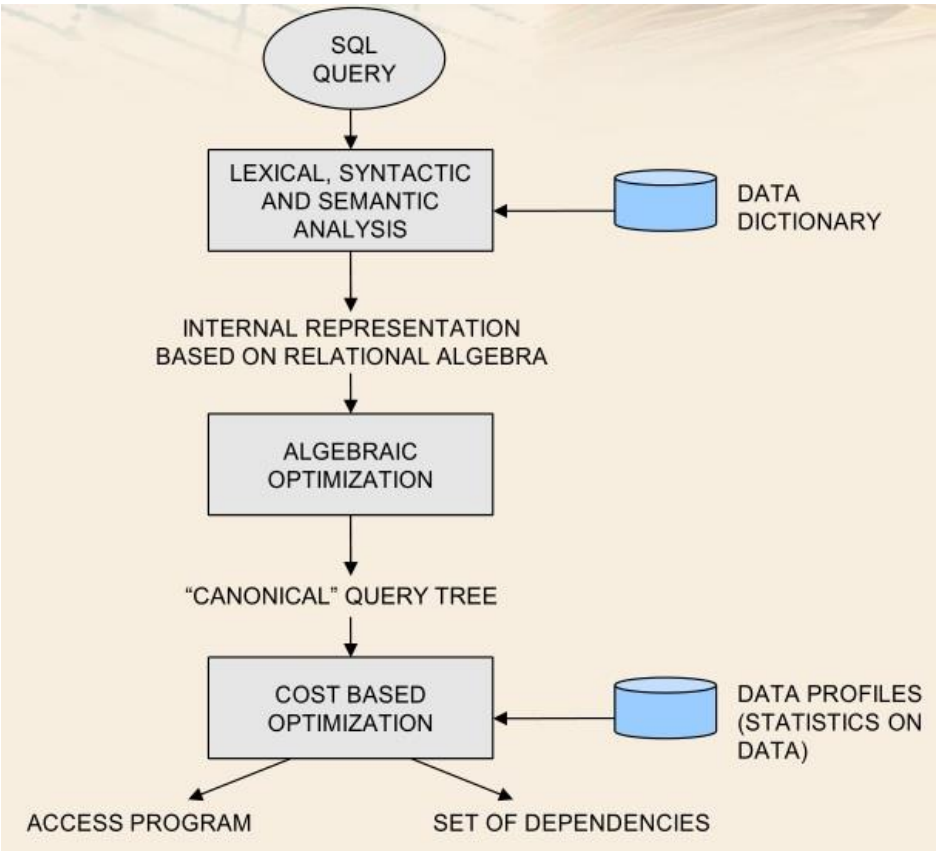
E' proprio questa proprietà fondamentale che ha garantito la diffusione e il largo utilizzo (ancora oggi) dei DBMS relazionali. Modificare le applicazioni dei dati comporta infatti un costo elevato [☹]e il rischio di introdurre errori [☹].

L'ottimizzatore genera *automaticamente* un **piano di esecuzione** della query (ovvero qualsiasi operazione DML che richieda un piano di esecuzione per l'accesso ai dati: interrogazione, aggiornamento, cancellazione, inserimento).

© Proprietà riservata dell'autore - Digitalizzazione e distribuzione a cura del CENTRO APPUNTI - Corso Luigi Einaudi, 55 - Torino / Pagina 89 di 598

Funzionamento dell'ottimizzatore

All'interno dell'ottimizzatore vengono eseguiti diversi passi.



Il primo passo consiste nell'andare a ricevere l'interrogazione SQL e trasformarla in un formato interno di rappresentazione (basato sull'algebra relazionale) più comodo da manipolare. Per far ciò è necessario comprendere ciò che è scritto nell'istruzione, ovvero effettuare l'analisi lessicale, sintattica e semantica di quest'ultima.

Analisi lessicale, sintattica e semantica

- L'analisi **lessicale** consiste nell'andare ad analizzare l'istruzione al fine di rilevare errori lessicali (Es. parole chiave scritte in maniera errata); per far questo è ovviamente necessario capire quando una determinata parola chiave deve essere presente nella struttura di una interrogazione.

- L'analisi **sintattica** consiste nell'andare ad analizzare l'istruzione al fine di rilevare errori di sintassi, ovvero errori nella grammatica del linguaggio SQL (Es. errore nello scrivere prima `FROM` poi `SELECT`).

Questa ottimizzazione dovrebbe anche permettere di notare espressioni **equivalenti** ed eliminare le differenze tra formulazioni differenti della stessa query (Es. due interrogazioni che forniscono gli stessi risultati, una scritta con il Join l'altro con l'`IN`). In passato questo passo non veniva eseguito, per cui una stessa interrogazione scritta in maniera differente forniva prestazioni di esecuzione differenti, ad oggi invece gli ottimizzatori più performanti riescono ad eseguire questo tipo di operazione.

Si osservi che è fondamentale che le query in questione vadano comunque ad operare sullo stesso numero di attributi e tabelle per essere considerare equivalenti, se invece si va ad esempio ad eseguire una query corretta ma che utilizza inutilmente il join con una tabella in più, è evidente che tale operazione peserà in termini di prestazioni e le due query non saranno considerate equivalenti.

L'ottimizzazione algebrica è generalmente indipendente dalla distribuzione dei dati e fornisce quindi come output un query tree in forma "**canonica**", ovvero una forma migliore di quella di partenza.

A questo punto, l'albero in forma canonica viene processato dall'ottimizzatore basato sui costi.

Ottimizzazione basata sui costi

L'ottimizzazione basata sui costi permette di selezionare il miglior piano di esecuzione della query, valutando i costi di esecuzione. In particolare seleziona:

- Il miglior **metodo di accesso** per ogni tabella (Es. usare un indice, usare uno scan sequenziale, etc.);
- Il miglior **algoritmo**, tra le alternative disponibili, per realizzare ogni operatore relazionale presente (Es. `JOIN`, `SELECT`, `GROUP BY`, etc.).

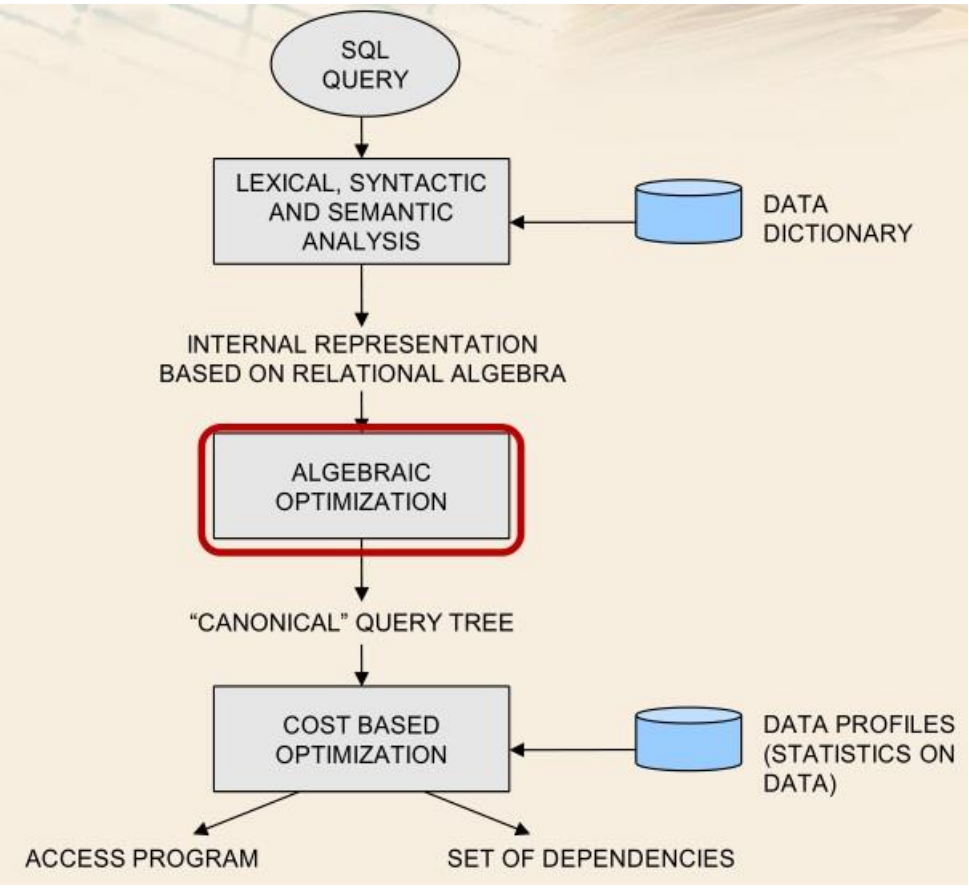
Tale scelta è fatta in funzione delle eventuale strutture dati accessorie disponibili (Es. indici), della distribuzione dei dati (Es. tabelle piccole), etc.

Si utilizza quindi un modello basato sui costi per definire gli algoritmi di esecuzione delle operazioni e i metodi di accesso ai dati. Questo significa che l'ottimizzatore fa una stima del costo di esecuzione delle diverse soluzioni che **dipende da** come sono fatti i **dati** contenuti in quel momento nel database. Da questo deriva la proprietà dell'ottimizzatore di **adattarsi dinamicamente** alla struttura del database e al suo contenuto (Es. se viene sostituito un indice con un altro, il sistema rileva il cambiamento e verifica se è conveniente o meno usare il nuovo indice inserito in funzione dei costi di utilizzo).

Il passo finale è la generazione del codice (quindi istruzioni e direttive) che implementa la migliore strategia scelta.

Ottimizzazione Algebrica

Si prende ora in esame la porzione dell'ottimizzatore che si occupa di effettuare l'ottimizzazione algebrica.



L'**ottimizzatore algebrico** riceve in ingresso una rappresentazione in algebra relazionale della query da eseguire e fornisce in uscita una nuova rappresentazione ottimizzata (cioè migliorata da un punto di vista delle prestazioni) in algebra relazionale della stessa query.

Tale modulo è basato su alcune proprietà fondamentali degli operatori dell'algebra relazionali, in particolare su quelle che vengono chiamate **trasformazioni di equivalenza**.

Due espressioni relazionali (quindi due query tree) si dicono **equivalenti** se producono lo stesso risultato indipendentemente dall'istanza del database considerata, ovvero indipendentemente dalle tuple presenti nella base di dati.

Le trasformazioni più interessanti sono quelle che:

- Permettono di ridurre la dimensione dei risultati intermedi che devono essere salvati in memoria per le elaborazioni successive; ad esempio se si deve fare un `JOIN` o una `SELECT` è meglio eseguire prima la `SELECT`, infatti il risultato intermedio che viene memorizzato eseguendo la selezione è più piccolo rispetto a quello che bisognerebbe memorizzare se si eseguisse prima il `JOIN`.

3. Anticipazione di una selezione rispetto ad un `JOIN` (**push della selezione**):

$$\sigma_F (E_1 \bowtie E_2) \equiv E_1 \bowtie (\sigma_F (E_2))$$
F is a predicate on attributes in E_2 only

Tale trasformazione indica che se si ha un predicato di selezione *F* che viene applicato al risultato del `JOIN` di due espressioni (E_1 ed E_2) e che coinvolge attributi di sola una delle due espressioni (ad esempio E_2), è possibile anticipare l'esecuzione della selezione su tale espressione, rispetto al `JOIN`.
In questo modo il `JOIN` lavorerà su E_1 ed un risultato più piccolo rispetto ad E_2.

Nel caso in cui la selezione avesse un predicato che interessasse sia E_1 che E_2, sarebbe possibile utilizzare prima la trasformazione di atomizzazione della selezione (#1) e solo successivamente si potrebbe applicare la trasformazione di push della selezione (#3).

4. **Anticipazione della proiezione rispetto al `JOIN`:**
Si consideri una situazione in cui si ha una proiezione rispetto agli attributi *L* applicata al `JOIN` con predicato *p* tra due espressioni (E_1 ed E_2). L'insieme *L* di attributi potrà avere attributi appartenenti ad E_1 ed attributi appartenenti ad E_2, per questo si può definire L_1 il sottoinsieme di *L* contenente solo gli attributi che appartengono ad E_1 (ovvero $L_1 = L - Schema(E_2)$) e analogamente L_2 come il sottoinsieme di *L* contenente solo gli attributi che appartengono ad E_2 ($L_2 = L - Schema(E_1)$).
A questo punto, per poter anticipare la proiezione sarà necessario fare in modo che le espressioni E_1 ed E_2, dopo essere state sottoposte alla proiezione, abbiano ancora degli attributi in comune, per questo motivo si può definire *J* come l'insieme degli attributi che sono necessari per valutare il predicato di JOIN (*p*).
In definitiva si può applicare la seguente trasformazione:

$$\pi_L(E_1 \bowtie_p E_2) \equiv \pi_L ((\pi_{L1,\,J}(E_1)) \bowtie_p (\pi_{L2,J}(E_2)))$$
- L1 = L - Schema(E_2)
- L2 = L - Schema(E_1)
- J = set of attributes needed to evaluate join predicate p

Per cui la proiezione rispetto agli attributi *L* applicata al `JOIN` con predicato *p* tra due espressioni (E_1 ed E_2) è equivalente alla proiezione fatta rispetto agli attributi *L* del `JOIN`, con predicato *p*, tra:
   - La proiezione rispetto agli attributi L_1 e *J* di E_1;
   - La proiezione rispetto agli attributi L_2 e *J* di E_2.