Tecnologie Delle Basi di Dati M

Antonio Davide Calì

1 maggio 2014

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

Indice

Parte I
Progetto fisico e tuning del Database

La valutazione del progetto di un Database avviene tramite la misurazione delle prestazioni del *dmbs* rispetto alle interrogazioni ed alle operazioni di modifica del Database tipicamente utilizzate. È possibile migliorare le prestazioni *a priori* tramite i tre tipi di progettazione (concettuale, logica e fisica) oppure *a posteriori* tramite il tuning di parametri e/o oggetti del database. Se mentre la progettazione concettuale e logica sono state trattate in corsi precedenti, per

progettazione fisica si intende *mappare* uno schema logico sull'hardware che abbiamo a disposizione: tutto ciò, nella realtà, è un problema molto complicato poichè richiede il tipico problema, che non tratteremo, di come distribuire i dati su più macchine e architetture con I/O diversi il che potrebbe richiedere una frammentazione dei dai orizzontale o verticale per migliorarne il parallelismo.

La chiave per un buon progetto fisico è ottenere una descrizione il più possibile accurata del carico di lavoro atteso. Il **workload** fa proprio questo, include dunque:

- Un elenco di query con la rispettiva frequenza di esecuzione
- Un elenco di transazioni (operazioni di modifica) con la rispettiva frequenza di esecuzione
- Un obiettivo di prestazione per ciascun tipo di query e/o operazione di modifica

In pratica nella desrcizione del workload sono presenti le principali operazioni che avvengono sul database, il che, può farci capire, come sia utile ottenre una buona performance su query che vengono effettuate frequentemente piuttosto che query che avvengono di rado. Per ogni *query* nel workload occorre conoscere quali relazioni sono interessate, quali attributi vengono mantenuti (proiezione), quali attributi sono interessati da predicati (quali selezione e join), quale è la selettiva di ogni predicato (questo parametro in realtà è un parametro ignoto poichè se il database è vuoto non si può sapere a priori la selettività di un predicato, ecco perchè ci si basa su valori di default),... Invece, per ogni *transazione* nel workload occorre conoscere che tipo di modifica viene realizzata (se *insert*, *delete* o *update*), quali relazioni sono interessate, quali attributi sono interessati da predicati (selezione/join), quale è la seleittività di ogni predicato, quali sono i campi che vengono modificati,...

Un'altra scelta da dover effettuare riguardano il progetto fisico e il tuning di esso: una prima domanda da porsi riguarda quali indici creare. Ricordiamo che un indice può velocizzare alcune query/update, ma tipicamente rallenta le transazioni (poichè richiede la modifica anche dell'indice stesso) dunque su quali relazioni/attributi conviene costruire un indice? L'indice deve essere *clustered*? Deve essere *denso* o *sparso*?

Ci si può chiedere anche se occorre modificare lo schema concettuale per migliorare le prestazioni: si potrebbe pensare a schemi di normalizzazione alternativi, o alla denormalizzazione, o ancora al partizionamento verticale/orizzontale, o a viste materializzate, ... Si ricordi la normalizzazione e la denormalizzazione: potrebbe essere successo che due tabelle che sono state create altro non siano che la conseguenza di una normalizzazione di tipo 3; se una query richiede spesso dati da entrambe le tabelle potrebbe essere dunque conveniente *denormalizzare* riunendo tali dati in un'unica relazione (con la conseguenza di creare ridondanza) per evitare di appliccare il join alle relazioni per riuscire ad ottenere i dati voluti. Il *partizionamento verticale* è molto simile ad una normalizzazione: dati tutti gli attributi di una relazione, se essi non sono "effettivamente" utilizzati

assieme, potrebbe convenire creare due relazioni piuttosto che una singola, in cui i vari attributi vengono appunto partizionati nelle due tabelle. Il *partizionamento orizzontale* invece implica creare due tabelle con gli stessi attributi, ma che contengano dati diversi: caso tipico avviene quando si vogliono tenere gli "storici" e i valori attuali di una relazione; ciò potrebbe portare a pensare a partizionare effettivamente i dati in due tabelle identiche in cui inserire nella prima solo gli "storici" mentre nella seconda solo i dati attuali. Parlare di *viste materializzite* potrebbe sembrare un paradosso poichè per definizione le viste sono entità "virtuali" che si appoggiano alle tabelle fisiche: richiederne la loro materializzazione, e dunque creare effettivamente dei dati fisici, potrebbe essere un fattore utile per la performance, infatti alcuni *dbms* potrebbero non essere in grado di "riscrivere" determinate query (si veda il precedente capitolo) per risolvere in maniera ottimale le viste e dunque il loro unico modo di calcolare i risultati è calcolare il risultato della vista e poi appliccarlo all'interno della query. Materializzare una vista, dunque, se utilizzata molte volte ed essa è soggetta a pochi cambiamenti, può risultare comodo poichè parte del lavoro risulta essere già svolto e la query effettivamente andrebbe ad interrogare una tabella esistente nel database.

Occorre riscrivere anche alcune query e/o transazioni per migliorare le prestazioni? Si noti che non tutti i *dbms* sono in grado di applicare la "riscrittura" delle query, come ad esempio per effettuare l'*unnesting* delle subquery, e quindi è richiesto al DBA l'arduo compito di farlo in maniera manuale. Un altro caso tipico, ormai risolto con le nuove versioni dei *dbms*, era l'ordine di accesso alle relazioni: cioè l'ordine in cui si scrivevano le relazioni nella clausola FROM era l'ordine con cui si accedeva effettivamente alle relazioni (ad esempio $FROM\ R1,\ R2 \neq FROM\ R2,\ R1$) e dunque poteva portare ad avere performance differenti.

Si noti infine che il tutto si complica in caso di scenari distribuiti.

1 Tuning

Il *tuning* si rende necessario poichè, spesso, le condizioni al contorno cambiano. È dunque necessario rivedere alcune delle scelte progettuali effettuate all'inizio:

- Nuove query/transazioni
- Modifica delle frequenze delle query/transazioni
- Upgrade/downgrade dell'hardware
- Modifica/aggiornamento del *dbms*
- ...

Possiamo utilizzare **5 principi fondamentali** per effettuare il tuning:

1. *"Think globally, fix locally"* - Non limitarsi all'ottimizzazione di singole query, poichè bisogna sempre pensare che migliorando magari una singola query si inficia la performance a livello globale del database.

2. *"Partitioning breaks bottlenecks"* - Partizionare nel tempo, nello spazio o in risorse. Si pensi al collo di bottiglia come se fosse presente una coda nel traffico, per poter migliorare la situazione esistono prettamente due soluzioni: cercare di velocizzare i veicoli in modo che scorrano più velocemente oppure suddividere il traffico su più percorso. La prima soluzione è assimilabile all'ottimizzazione delle query, la seconda invece è al miglioramento delle performance partizionando il lavoro. Il partizionamento può avvenire in tre modi: attraverso risorse, quindi magari avere a disposizione più dischi su cui far lavorare il database, oppure partizionando lo spazio o ancora partizionando il tempo. Un tipico esempio di partizionamento nel tempo riguarda le transazioni che hanno un'interazione con l'utente: si immagini infatti che una transazione, una volta partita, ad un certo punto richieda un input da parte dell'utente, il quale può impiegare un tempo impredicibile nel rispondere; il problema nasce poichè la transazione avrà acquisito dei lock su delle risorse, e se deve attendere la risposta dell'utente per poter rilasciarli, blocca anche tutte le altre transazioni che vorrebbero accedere a quelle determinate risorse: eco che la soluzione potrebbe richiedere una partizione temporale, cioè nella prima parte la transazione richiede i lock, ma poco prima di chiedere l'input dell'utente li rilascia per poi riguadagnarli a risposta avvenuta.

3. *"Start-up costs are high, running costs are low"* - I costi iniziali per determinate operazioni sono elevati, mentre i costi di esecuzione tendono ad essere bassi. Ad esempio il caso di letture sequenziali che richiede un primo tempo di seek (alto) per poi procedere col solo tempo di trasferimento (basso), o ancora la compilazione di query che richiede un'ottimizzazione la prima volta, o ancora la connessione al database da parte di applicazioni (tempo alto) rispetto a mantenere la connessione attiva per eseguire le query successive (tempo basso) quindi richiedendo solo una prima connessione e mantendola attiva per una certa durata di tempo piuttosto che connettersi ogni volta al database,...

4. *"Render unto server what is due unto server"* - Cercare di capire cosa va fatto sul server e cosa va (può) essere fatto sul client: sviluppare quindi le applicazioni per ridurre il carico sui server. Ad esempio un check semantico può avvenire anche su lato client, evitando di inviare query errate al server, o ancora evitare il polling da parte del client sul server per sapere se magari il database ha avuto una modifica, affidando piuttosto il compito al server di notificare il client in caso di un determinato evento (attraverso dei trigger ad esempio).

5. *"Be prepared for trade-offs"* - Bisogna essere consci che vanno fatti dei compromessi. "You want speed: how much are you willing to pay?"

Il *problema del progetto fisico* può essere visto come un problema di ottimizzazione in cui il costo di ogni query/transazione viene modificato dalla presenza/assenza di un indice. È però evidente che lo spazio delle possibili soluzioni è

enorme: utilizziamo dunque un approccio euristico incrementale. In primo luogo vediamo quali indici potrebbero migliorare le prestazioni delle query/transazioni più importanti, quindi vediamo se l'aggiunta di ulteriori indici può migliorare la soluzione. È possibile anche affidarsi a strumenti automatici, che discuteremo a breve, che si basano su analisi "*what-if*" i quali analizzano il database simulando la presenza o l'assenza di determinati indici. Si noti che esiste un unico sistema *dbms* (SQL SERVER) che ha sviluppato qualcosa di significativo sul *self tuning*, in cui è il dbms stesso che osservando l'andamento di query con la loro frequenza e le performance ottenute, riesce a riconfigurarsi da solo creando e/o eliminando determinati indici per migliorarsi da solo.

1.1 Linee guida

Vediamo di seguito alcune linee guida per un buon *tuning* del nostro database.

Nella creazione di un indice bisogna come prima cosa pensare di creare indici che velocizzino più di una query, evitando di creare indici inutili. Per la scelta dell'attributo da indicizzare possiamo pensare di utilizzare gli attributi presenti nella clausola WHERE delle query (sono ottimi candiditati a chiavi di ricerca). I predicati di uguaglianza suggeriscono l'uso di indici *hash*, invece i predicati di range suggersicono l'uso di B+tree. Infine ricordiamo che il *join index nested loop* funziona meglio con gli indici *hash*.

Possiamo anche pensare di utilizzare chiavi multiple in un indice, ad esempio se siamo in presenza di predicati WHERE su più attributi di una relazione. Inoltre ci potrebbe permettere di sfruttare piani di accesso INDEX ONLY (che permettono la risoluzione senza dover accedere ai dati). **NB** Dobbiamo far attenzione all'ordine degli attributi in caso di predicati di range. Ad esempio se avessi la query col predicato WHERE A=5 AND B BETWEEN 1 AND 10, avere un indice B+tree su (A,B) risulta diverso che avere un indice su (B,A): se infatti nel primo, grazie al predicato A=5 riesco a limitare di molto il sotto insieme di tuple su cui poi controllare il B, diverso risulta sul secondo indice in quanto il numero di tuple residue del predicato B BETWEEN 1 AND 10 si assume essere molto più grande, sulle quali poi controllare il predicato di A.

Un'altro punto di cui tener conto è la *clusterizzazione* di un indice: le query di range sono quelle che beneficiano maggiormente dalla clusterizzazione di un indice, ma anche le ricirche su attributi non chiave traggono vantaggio da indici *clustered*. Infine ricordiamoci che i piani di accesso INDEX ONLY non necessitano invece che l'indice sia *clustered* (non devono accedere ai dati, quindi che l'indice risulti clustered è inutile, di conseguenza sarebbe bene rendere clustered un altro indice che effettivamente ne ha bisogno, ricordando che solo un indice può risultare clustered per una tabella).

1.1.1 Selezione degli indici

Prendiamo la seguente query

```
SELECT E.name, D.mgr
FROM Employees as E, Departments as D
```

```
WHERE D.name='Toy' AND E.dno=D.dno
```

Quali indici usare?
Un indice *hash* su D.name (risolverebbe il predicato del WHERE), oppure un indice su E.dno (aiuterebbe il join) o ancora un indice su D.dno (aiuterebbe il join)?
Ciò di cui dobbiamo tener conto, è capire quale delle due relazioni risulterà interna e quale esterna nella risoluzione del join attraverso l'index nested loop. Notiamo che un possibile piano di accesso sarebbe recuperare i dipartimenti su D.name che risolvono *D.name='toy'* e utilizzare successivamente un *index nested loop* su E (che richiede un indice su E.dno e che risulterà quindi la relazione interna): quindi un indice *hash* su E.dno è una buona soluzione (collegato anche ad un indice su D.name per avere in breve i dati su cui effettuare dopo il join).

Modifichiamo la query come segue

```
SELECT E.name, D.mgr
FROM Employees as E, Departments as D
WHERE D.name='Toy' AND E.dno=D.dno AND E.age=25
```

Abbiamo introdotto un predicato di WHERE anche per la relazione E (E.age=25). Ora quali indici possiamo utilizzare? Su D.name, E.age, E.dno o D.dno?
Se esistesse già l'indice su E.age potrebbe essere inutile creare l'indice su E.dno (ma richiederebbe l'indice su D.dno per rendere D relazione interna e grazie ad esso poter poi effettuare l'index nested loop).

Modifichiamo nuovamente la query come segue

```
SELECT E.name, D.mgr
FROM Employees as E, Departments as D
WHERE E.sal BETWEEN 10000 AND 20000
        AND E.hobby='stamps' AND E.dno=D.dno
```

Ancora una volta ci chiediamo quali indici possiamo utilizzare: E.sal, E.hobby, E.dno o D.dno?
Tipicamente la scelta cadrà sull'indice più selettivo tra E.sal e E.hobby con l'aggiunta di D.dno (per effettuare il solito index nested loop). E se la selettività non fosse nota? Ogni *dbms* ha dei valori di default per i predicati (ad esempio DB2 per il predicato di uguaglianza ha come valore di default $\frac{1}{10}$).

1.1.2 Indici clustered

Quando creo un indice, conviene renderlo *clustered*? Ricordiamo che solo un indice può essere clustered per ogni tabella.

Prendiamo la seguente query

```
SELECT E.dno
FROM Employees as E
WHERE E.age>40
```

L'indice su E.age (per forza un B+tree poichè predicato su range) dovrebbe essere *clustered*? Se ci sono molti "ultra-quarantenni" l'indice non è utile poichè

l'indice serve per risparmiare i costi, ma se esistono molti ultra quarantenni l'accesso sequenziale ai dati potrebbe risultare migliore. E se ci sono ad esempio solo il 10% di "ultra-quarantenni", l'indice *clustered* potrebbe effettivamente velocizzare l'accesso.

Cambiamo query

```
SELECT E.dno, count(*)
FROM Employees as E
WHERE E.age>30
GROUP BY E.dno
```

L'indice su E.age (di nuovo un B+tree) è utile? Dipende dalla selettività del predicato sull'attributo *age* poichè se la selezione è bassa, le tuple residue risultano molte e l'accesso sequenziale potrebbe comunque convenire. Un'alternativa potrebbe essere anche un indice sull'attributo *dno*: in questo caso l'indice dovrebbe essere *clustered* (alternativamente l'ottimizzatore sceglierebbe di ordinare la relazione E in base all'attributo *dno*) poichè mi aiuta ad effettuare il GROUP BY.

Prendiamo un'altra query

```
SELECT E.dno
FROM Employees as E
WHERE E.hobby='stamps'
```

L'indice sull'attributo *hobby* deve essere clustered? Dipende nuovamente dalla selettività del predicato su *hobby*. Se il predicato è poco selettivo e quindi ho molte tuple, se l'indice risulta clustered si riesce ad avere il risultato in poco tempo, se invece il predicato è molto selettivo e quindi le tuple risultano poche che sia clustered o unclustered ha poca importanza (si noti che in questo caso rispetto a quanto detto precedentemente, il predicato è di uguaglianza e non di range).

E nel seguente caso?

```
SELECT E.dno
FROM Employees as E
WHERE E.eid=552
```

Il fatto che l'indice sia *clustered* o *unclustered* (su E.eid) è irrilevante poichè il predicato WHERE riguarda l'attributo chiave.

Consideriamo ancora una nuova query

```
SELECT E.name, D.mgr
FROM Employees as E, Departments as D
WHERE D.name='Toy' AND E.dno=D.dno
```

Gli indici su D.name e E.dno devono essere clustered? Probabilmente ci saranno pochi dipartimenti che soddisfano il predicato, quindi l'indice può essere *unclustered*: viceversa l'indice che utilizzeremoper risolvere il join attraverso l'*index nested loop* dovrebbe essere *clustered.*

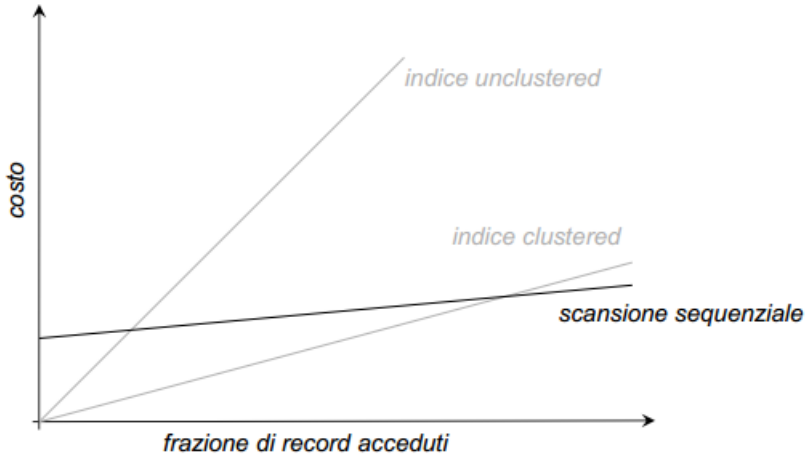
Analizziamo un'ultima query

```
SELECT E.name, D.mgr
FROM Employees as E, Departments as D
WHERE E.hobby='stamps' AND E.dno=D.dno
```

In questo caso quale relazione dovrebbe essere esterna? E quali indici risultano utili?
Un *sort-merge join* potrebbe sfruttare un B+tree *clustered* su D.dno; inoltre se il predicato su E.hobby è selettivo un indice (clustered) sull'attributo *hobby* potrebbe essere utile.

In generale il costo dell'accesso ai record di una tabella segue l'andamento mostrato in figura (Fig. 1).

Figura 1:



1.1.3 Indici multi-attributo

Prendiamo la seguente query

```
SELECT E.eid
FROM Employees as E
WHERE E.age BETWEEN 20 AND 30
        AND E.sal BETWEEN 3000 AND 5000
```

Un indice B+tree *clustered* sugli attributi (E.age, E.sal) potrebbe essere utile. Mentre un indice sulla coppia (E.sal, E.age)? Potrebbe essere anch'esso utile. Dipende tutto dalla selettività dei predicati.

E nel seguente caso?

```
SELECT E.eid
FROM Employees as E
WHERE E.sal BETWEEN 3000 AND 5000
        AND E.age=25
```

L'indice su (E.age, E.sal) conviene decisamente di più poichè le tuple residue lasciate da E.age=25 risulteranno molto minori.

1.1.4 Indici per INDEX ONLY

Prendiamo la seguente query

```
SELECT D.mgr
FROM Departments as D, Employees as E
WHERE D.dno=E.dno
```

Se abbiamo un indice denso sull'attributo E.dno possiamo usarlo per un *index nested loop* (quindi con la relazione E interna). Il piano risulta essere INDEX ONLY poichè una volta controllata l'esistenza di un match nel join, basterà ritornare il manager del dipartimento e quindi non ha alcun senso che l'indice sia *clustered* infatti nessun record della relazione E deve essere recuperato.

Se invece modifichiamo la query in

```
SELECT D.mgr, E.eid
FROM Departments as D, Employees as E
WHERE D.dno=E.dno
```

in cui vengono richiesti anche dati della relazione E , se abbiamo un indice denso, come prima, su E.dno lo possiamo usare per un *index nested loop* (immaginiamo E interna e D esterna) con però l'accortezza che l'indice deve essere *clustered* poichè altrimenti si avrebbero troppi accessi casuali. Se abbiamo un B+tree *denso* sulla coppia di attributi (E.dno, E.eid) possiamo ancora usare un piano di accesso INDEX ONLY e nuovamente non è necessario che l'indice risulti *clustered*. Un indice hash andrebbe bene? No, infatti per avere un indice hash occorre avere i dati per effettuare la ricerca, cioè occorre avere la coppia (E.dno, E.eid), ma non avendo E.eid (poichè è il risultato della query) non posso usarlo, mentre l'indice B+tree sì poichè basta avere solo un prefisso (in questo caso E.dno) della chiave con cui è stato generato.

Analizziamo la seguente query

```
SELECT E.dno, COUNT(*)
FROM Employees as E
GROUP BY E.dno
```

Se abbiamo un indice su E.dno lo possiamo utilizzare per un piano di accesso INDEX ONLY semplicemente contando quante tuple ho.

Modificando la query come segue

```
SELECT E.dno, COUNT(*)
FROM Employees as E
WHERE E.sal=10000
GROUP BY E.dno
```

risulta più conveniente avere un indice sulla coppia (E.sal, E.dno) o su (E.dno, E.sal)? La prima scelta risulta essere migliore poichè il numero di

tuple residue che poi serviranno per il GROUP BY risulteranno decisamente inferiori. Ma è meglio un indice *hash* o B+tree? In questo caso il B+tree risulta conveniente poichè dobbiamo successivamente ordinare (per effettuare il group by) per l'attributo E.dno.

Controlliamo una nuova query

```
SELECT E.dno, MIN(E.sal)
FROM Employees as E
GROUP BY E.dno
```

e ci chiediamo nuovamente se risulta più conveniente avere un indice sulla coppia (E.sal, E.dno) o su (E.dno, E.sal). In questo caso un indice B+tree sulla coppia (E.dno, E.sal) ci darebbe immeditamente il risultato (poichè l'ordinamento dovuto al group by permetterebbe di avere subito l'informazione sul minimo, cioè la prima tupla di ogni E.dno), mentre il piano di accesso che utilizzi l'indice (E.sal, E.dno) richiederebbe anch'esso un B+tree ma risulterebbe comunque meno efficiente.

Analizziamo un ultima query

```
SELECT AVG(E.sal)
FROM Employees as E
WHERE E.age=25
        AND E.sal BETWEEN 3000 AND 5000
```

Conviene avere un indice sulla coppia (E.sal, E.age) o su (E.age, E.sal)? In questo caso un indice B+tree su (E.age, E.sal) ci darebbe immeditamente il risultato, mentre il piano di accesso che utilizza l'indice (E.sal, E.age) richiederebbe anch'esso un B+tree ma sarebbe comunque meno efficiente.

1.2 Tuning del Database

Come precedentemente anticipato il tuning del database è fondamentale. È richiesto tutte quelle volte in cui le statistiche del database variano: ciò suggerisce che si rende anche necessario aggiornare le statistiche, in quanto in caso di aggiornamenti e modifiche, le scelte effettuate sulla costruzioni di indici, query o quant'altro potrebbe essersi inficiata e occorre dunque la modifica delle scelte effettuate.

Il tuning riguarda diversi punti che andremo brevente ad elencare.

1. **Tuning degli indici**: che può avvenire attraverso la rivalutazione delle scelte in base a statistiche modificate, dall'aggiornamento delle statistiche degli indici e del database e dalla riorganizzazione periodica gli indici.

2. **Tuning dello schema concettuale**: che può avvenire attraverso la denormalizzazione, la decomposizione verticale, la decomposizione orizzontale e ad un'evenutale creazione di viste.

3. **Tuning di query (e viste)**: che può avvenire attraverso l'uso di UNION piuttosto che del predicato OR, oppure attraverso l'eliminazione della clau-

sola DISTINCT (poichè magaril il dbms non riesce a riscrivere automaticamente la query eliminandolo), o alla sostituzione delle sub-query con dei join (sempre per lo stesso motivo che il dbms non sappia come riscrivere le query), o con l'eliminazione delle tabelle temporanee, o con la riscrittura di predicati con condizioni aritmetiche (ad esempio conviene avere $E.age = 2 \cdot D.age$ piuttosto che $\frac{E.age}{2} = D.age$) o infine con selezioni con valori NULL.

4. **Impatto della concorrenza**: che può avvenire riducendo il livello di *isolation* delle query (magari alcune transazioni permettono la dirty read, e quindi richiedono un livello di isolation più bassa).

2 Strumenti Automatici

Quasi tutti i *dmbs* commerciali forniscono degli strumenti automatici per la progettazione fisica e il tuning del database. Alcuni esempi possono essere DB2 design advisor, SQL Server database tuning advisor, Oracle access advisor, ... Tipicamente si basano sull'analisi "*what-if*?" in cui si usa l'ottimizzatore per valutare l'impatto di una possibile scelta. Ogni sistema ha poi una varietà di sturmenti per la gestione fisica dei dati, l'analisi dei piani di accesso, ecc...

Una raccomandazione di cui tener conto: i suggerimenti forniti dagli *advisor* si rifersicono unicamente al *workload* fornito. Non è possibile, cioè, trarre considerazioni generali che valgano per workload diversi da quello fornito. Questo significa che non è mai opportuno "sovra-ottimizzare" un database (cioè renderlo perfettamente ottimo con i dati forniti con la conseguenza che alla prima variazione di statistiche tutto il lavoro fatto diventa inutile): occorre comunque che il Database Administrator conosca quali siano le conseguenze di ciascuna azione suggerita dall'advisor.

In DB2 le informazioni statistiche su tabelle e indici sono fondamentali per permettere all'ottimizzatore di operare scelte accurate. Il comando messo a disposizione per collezionare le statistiche è *RUNSTATS*. Esistono però molte varianti tra cui la seguente

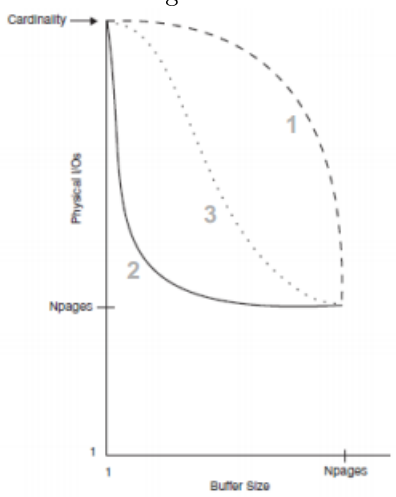
```
RUNSTATS ON TABLE MySchema.TableName WITH DISTRIBUTION
        ON ALL COLUMNS AND DETAILED INDEXES ALL
```

in cui la forma WITH DISTRIBUTION forza DB2 a colleezionare statistiche dettagliate sulle distribuzioni dei valori delle varie colonne (vedasi *istogrammi* e valori frequenti). La forma DETAILED INDEXES ALL invece genera informazioni utili per capire il costo di una scansione via indice: in particolare il sistema genera una lista di 11 *PAGE_FETCH_PAIRS* in cui ogni coppia ha la forma (*buffer_size*, *#I/O*) e stima quante operazioni I/O (fisiche) saranno necessarie eseguire per effettuare una scansione completa avendo a disposizione un certo numero di buffer (buffer_size).

Alcuni esempi di curve *PAGE_FETCH_PAIRS* "tipiche", facendo riferimento alla figura (Fig. 2), sono le seguenti:

1. *Clustering* praticamente nullo
2. Buon *clustering*, avendo un buffer size modesto si evitano I/O inutili
3. Situazione intermedia, assunta dall'ottimizzatore in assenza di statistiche dettagliate

Figura 2:



È possibile controllare se la *riorganizzazione dei dati* risulta necessaria: il comando messo a disposizione è **REORGCHK**, e vediamolo applicato

```
REORGCHK ON TABLE MySchema.TableName
```

Il comando fornisce alcuni indicatori utili per capire se una *table* deve essere riorganizzata fisicamente (lo stesso vale per gli indici). Il comando esegue automaticamente anche RUNSTATS secondo la modalità presente in catalogo (cioè utilizza la stessa modalità di RUNSTATS che fornisce i dati già presenti nel catalogo).

In figura (Fig. 3) viene mostrato un esempio di informazioni fornite da REORGCHK, in cui vengono forniti tre indicatori con relativi valori di soglia (in caso il valore risulti minore o maggiore, a seconda dei casi, del valore di soglia viene emesso un WARNING).

F1 Percentuale di record in overflow (richiede che il valore sia $< 5\%$)

F2 Percentuale di spazio utilizzato nelle pagine allocate (richiede che il valore sia $> 70\%$)

F3 $\frac{NPAGES}{FPAGES}$ (richiede che il valore sia$> 80\%$) in cui NPAGES è il numero di pagine che contengono effettivamente una tupla, invece FPAGES è il numero di pagine totali assegnate alla tabella.

Figura 3:

```
db2 => REORGCHK ON TABLE B16884.TEST;

RUNSTATS in esecuzione....


Statistiche della tabella:

F1: 100 * OVERFLOW / CARD < 5
F2: 100 * (Utilizzo effettivo di spazio di pagine di dati) > 70
F3: 100 * (Pagine richieste / Numero di pagine totali) > 80

SCHEMA.NAME                     CARD     OV     NP     FP ACTBLK    TSIZE  F1  F2  F3 REORG
-------------------------------------------------------------------------------------------
Tabella: B16884.TEST
                               16384      0    497    497      -  1982464   0 100 100 ---
-------------------------------------------------------------------------------------------
```

In caso la riorganizzazione risulti necessaria, DB2 mette a disposizione i seguenti comandi:

1. Per riorganizzare i dati e gli indici:

   REORG TABLE MySchema.MyTable

2. Per riorganizzare solo gli indici

   REORG INDEXES ALL FOR TABLE MySchema.MyTable

2.1 Esempio

Mostriamo un esempio di ciò che abbiamo appena visto. Per generare una situazione in cui la riorganizzazione può rendersi necessaria generiamo una tabella con molte righe, eseguiamo su essa *REORGCHK*, poi cancelliamo buona parte delle tuple e rieseguiamo *REORGCHK*.

Generiamo intanto i dati creando la seguente tabella

```
CREATE TABLE TEST (
        A       INT,
        B       INT,
        C     CHAR(100)
)
```

in cui C ci serveo solo a "occupare" spazio.

Per generare i dati usiamo un *trigger* ricorsivo: DB2 supporta al massimo 16 livelli di ricorsione, ma non permette la "ricorsione multipla" (ovvero non possiamo avere più di un'azione di inserimento nel trigger).

Utilizziamo il seguente trigger

```
CREATE TRIGGER AUTO_INSERT
AFTER INSERT ON TEST
FOR EACH STATEMENT
WHEN ( 10000> (SELECT COUNT(*) FROM TEST) )
INSERT INTO TEST(A,B,C)
        SELECT MOD(3*A+1,20),MOD(3*B+1,50),C FROM TEST
```

Il trigger va creato dopo l'inserimento del primo record e prima dell'inserimento del secondo:

```
INSERT INTO TEST VALUES (1,10,' ')
INSERT INTO TEST VALUES (2,20,' ')
```

Siamo riusciti a generare così *16384* record ($16384 = 2 + 2 + 4 + 6 + 16 + \ldots + 8192$).

Eseguiamo quindi un RUNSTATS (in cui B16884 è lo schema in cui si trova la nostra relazione TEST)

```
RUNSTATS ON TABLE B16884.TEST
        WITH DISTRIBUTION ON ALL COLUMNS
```

e dunque effuttiamo un REORGCHK che mostra il risultato in figura (Fig. 4)

Figura 4:

```
db2 => REORGCHK ON TABLE B16884.TEST;

RUNSTATS in esecuzione....


Statistiche della tabella:

F1: 100 * OVERFLOW / CARD < 5
F2: 100 * (Utilizzo effettivo di spazio di pagine di dati) > 70
F3: 100 * (Pagine richieste / Numero di pagine totali) > 80

SCHEMA.NAME                     CARD     OV     NP     FP ACTBLK    TSIZE  F1  F2  F3 REORG
-------------------------------------------------------------------------------------------
Tabella: B16884.TEST
                               16384      0    497    497      -  1982464   0 100 100 ---
-------------------------------------------------------------------------------------------
```

Effettuiamo successivamente alcune cancellazioni utilizzando la seguente query

```
DELETE FROM TEST
WHERE B>3
```

e richiediamo le statistiche ottenendo il risultato mostrato in figura (Fig. 5), in cui si vuole evidenziare il valore F2 (in cui l'utilizzo effettivo è crollato al solo 26%), mentre si noti che F3 risulta essere ancora 100 (nonostante NP sia calato di una unità).

Figura 5:

```
Statistiche della tabella:

F1: 100 * OVERFLOW / CARD < 5
F2: 100 * (Utilizzo effettivo di spazio di pagine di dati) > 70
F3: 100 * (Pagine richieste / Numero di pagine totali) > 80

SCHEMA.NAME                     CARD     OV     NP     FP ACTBLK    TSIZE  F1  F2  F3 REORG
-------------------------------------------------------------------------------------------
Tabella: B16884.TEST
                                4290      0    496    497      -   519090   0  26 100 -*-
-------------------------------------------------------------------------------------------
```

ed infine riorganizziamo la tabella con il seguente statement

```
REORG TABLE B16884.TEST
```

che porta al risultato della seguente figura (Fig. 6)

Figura 6:

```
Statistiche della tabella:

F1: 100 * OVERFLOW / CARD < 5
F2: 100 * (Utilizzo effettivo di spazio di pagine di dati) > 70
F3: 100 * (Pagine richieste / Numero di pagine totali) > 80

SCHEMA.NAME                    CARD      OV      NP      FP ACTBLK    TSIZE  F1  F2  F3 REORG
-------------------------------------------------------------------------------------------
Tabella: B16884.TEST
                               4290       0     131     131      -   519090   0 100 100 ---
-------------------------------------------------------------------------------------------
```

in cui si vede che il valore di NP e FP è diminuito portandosi a raggiungere lo stesso valore, e F2 è tornato ad essere al 100%.

Informazioni dettagliate sul piano di accesso scelto dall'ottimizzatore possono essere mantenute in un insieme di tabelle chiamate **Explan Tables**. Le Explain tables possono essere esaminate mediante tool grafico del Command Editor (Visual Explain) oppure mediante semplici interrogazioni SQL.

In figura (Fig. 7) viene mostrato la richiesta di visualizzazione attraverso il Visual Explain del piano di accesso generato utilizzando il Command Editor di DB2.

Figura 7: