Progettazione fisica delle basi di dati

Introduzione

- Dopo il progetto ER, la tarduzione in relazionale, il raffinamento dello schema e la definizione delle viste, abbiamo gli schemi logico ed esterno della nostra base di dati.
- Il passo successivo consiste nella scelta degli indici, nel prendere decisioni sul *clustering*, e nel raffinamento degli schemi logico ed esterno (se necessario) per raggiungere gli obiettivi prestazionali
- Dobbiamo iniziare con la comprensione del carico di lavoro
  - Le interrogazioni più importanti e quanto spesso vengono usate
  - Gli aggiornamenti più importanti e quanto spesso vengono apportati
  - Le prestazioni desiderate per queste interrogazioni e per questi aggiornamenti

Decisioni da prendere

- Che indici dobbiamo creare?
  - Quali relazioni dovrebbero avere indici? Quali campi dovrebbero costituire la chiave di ricerca? Dovremmo costruire più indici?
- Per ciascun indice, che tipo di indice dovrebbe essere?
  - *Clustered*? *Hash*/albero?
- Dobbiamo apportare cambiamenti allo schema logico (e a quello concettuale)?
  - Considerare schemi normalizzati alternativi? (Ricordate, ci sono molte scelte per la decomposizione in BCNF, etc.)
  - Dovremmo "annullare" certi passi della decomposizione e indirizzarci verso una forma normale minore? (Denormalizzazione)
  - Partizionamento orizzontale, replicazione, viste...

Selezione degli indici per le operazioni di join

- Quando si considera una condizione di *join*:
  - gli indici *hash* sulle relazioni interne sono molto buoni per gli Index Nested Loops
    - dovrebbero essere *clustered* se la colonna di *join* non è la chiave della relazione interna, e bisogna leggere le tuple di quest'ultima
- Alberi B+ *clustered* sulle colonne di *join* sono buoni in caso di Sort-Merge

Abbiamo parlato degli indici per le interrogazioni su una sola tabella nel Capitolo 10

Esempio 1

```
SELECT I.inome, R.mgr
FROM Imp I, Rep R
WHERE R.rnome = 'Giocattoli' AND I.rnum = R.rnum
```

- Un indice *hash* su R.rnome supporta la selezione 'Giocattoli'
  - Ciò dato, un indice su R.rnum non è necessario
- Un indice *hash* su I.rnum ci permette di leggere le tuple (della relazione interna) di Imp per ciascuna tupla selezionata in Rep (nella relazione esterna)
- Che succederebbe se la clausola WHERE includesse: "... AND I.età = 25"?
  - Potremmo leggere le tuple di Imp usando un indice su I.età, poi fare un *join* con le tuple di Rep che soddisfano la selezione su rnome. Paragonabile alla strategia che usava l'indice su E.rnum
  - Quindi, se l'indice su I.età è già creato, questa interrogazione fornisce una motivazione molto debole per l'aggiunta di un indice su I.rnum

Esempio 2

```
SELECT I.inome, R.mgr
FROM Imp I, Rep R
WHERE I.sal BETWEEN 10000 AND 20000
        AND I.hobby = "Francobolli" AND I.rnum = R.rnum
```

- Chiaramente, Imp dovrebbe essere la relazione esterna
  - Ciò suggerisce la costruzione di un indice *hash* su R.rnum
- Che indice dovremmo costruire su Imp?
  - Potremmo usare un albero B+ su I.sal, OPPURE un indice su I.hobby. Solo uno di questi due è necessario, e quale dei due dipende dalla selettività delle condizioni
    - Come regola di massima, le condizioni con selezione di uguaglianza sono più selettive delle condizioni con selezione di intervallo
- Come indicato da entrambi gli esempi, la nostra scelta degli indici è guidata dai piani che ci aspettiamo vengano considerati dall'ottimizzatore per una interrogazione. Bisogna comprendere gli ottimizzatori!

Clustering e join

```
SELECT I.inome, R.mhr
FROM Imp I, Rep R
WHERE R.rnome = 'Giocattoli' AND I.rnum = R.rnum
```

- Il *clustering* è importante specialmente quando si accede alle tuple della relazione interna in un Index Nested Loop
  - L'indice su I.rnum dovrebbe essere *clustered*
- Supponiamo che la clausola WHERE sia invece:
- WHERE I.hobby = "Francobolli" AND I.rnum = R.rnum
  - Se molti impiegati collezionano francobolli, varrebbe la pena prendere in considerazione un *join* Sort-Merge. Un indice *clustered* su R.rnum sarebbe di aiuto
- Conclusione: il *clustering* è utile ogni volta che devono essere restituite molte tuple

Raffinare lo schema logico

- La scelta dello schema logico dovrebbe essere guidata dal carico di lavoro, oltre che a considerazioni sulla ridondanza
  - Possiamo indirizzarci verso uno schema 3NF piuttosto che a un BCNF
  - Il carico di lavoro potrebbe influenzare la scelta che facciamo nel decomporre una relazione in 3NF o in BCNF
  - Possiamo ulteriormente decomporre uno schema BCNF!
  - Potremmo denormalizzare (ossia annullare un passo di decomposizione), o potremmo aggiungere campi a una relazione
  - Potremmo considerare decomposizioni orizzontali
- Se tali modifiche sono fatte dopo che una base di dati è in uso, il che viene detto *evoluzione dello schema*, potremmo voler nascondere alcune di queste modifiche alle applicazioni definendo delle *viste*

Schemi di esempio

Contratti(cid, fid, gid, did, pid, qta, val)
Dep(did, budget, report)
Fornitori(fid, indirizzo)
Pezzi(pid, costo)
Progetti(gid, mgr)

- Ci concentreremo su Contratti, denotata come CFGDPQV. Sono dati i seguenti VI: GP _ C, FD _ P, C è la chiave primaria
  - Quali sono le chiavi candidate per CFGDPQV?
  - In quale forma normale si trova questo schema di relazione?

Decidere per la 3NF o per la BCNF

- CFGDPQV può essere decomposta in FDP e CFGDQV, ed entrambe le relazioni sono in BCNF (quale DF ci suggerisce di fare così?)
  - Decomposizione senza perdita, ma che non conserva le dipendenze
  - L'aggiunta di CGP permette di conservare le dipendenze
- Supponiamo che questa interrogazione sia molto importante:
  - Trovare il numero di copie Q di pezzi P ordinati nel contratto C
  - Richiede un *join* sullo schema decomposto, ma si può rispondere tramite una scansione della relazione originale CFGDPQV
  - Può indurci a decidere per lo schema 3NF CFGDPQV

Denormalizzazione

- Supponiamo che la seguente interrogazione sia importante:
  - il valore di un contratto è minore del budget del dipartimento?
- Per velocizzare questa interrogazione, potremmo aggiungere un campo budget B a Contratti
  - Questo introduce la DF D _ B rispetto a Contratti
  - Così Contratti non è più in 3NF
- Potremmo scegliere di modificare Contratti in tal senso, se l'interrogazione è sufficientemente importante, e non possiamo ottenere prestazioni adeguate in altro modo (ad esempio, aggiungendo indici o scegliendo uno schema 3NF alternativo)

Scelta delle decomposizioni

- Ci sono 2 modi per decomporre CFGDPQV in BCNF:
  - FDP e CFGDQV; *join* senza perdite ma non conserva le dipendenze
  - FDP, CFGDQV e CGP; conserva anche le dipendenze
- La differenza tra di esse è in realtà il costo del mantenimento della DF GP _ C
  - Seconda decomposizione: indice su GP nella relazione CGP
- Prima decomposizione

```
CREATE ASSERTION ControllaDip
CHECK (NaOT EXISTS (SELECT *
FROM PezzoInfo P, ContrattoInfo C
WHERE P.fid = C.fid AND P.did = C.did
GROUP BY C.gid, P.pid
HAVING COUNT(C.cid) > 1))
```

Scelta delle decomposizioni (segue)

- Valevano i seguenti VI:

  GP _ C, FD _ P, C è la chiave primaria

- Supponiamo che, inoltre, un dato fornitore applichi sempre lo stesso prezzo per un dato pezzo: FPQ _ V
- Se decidiamo di voler decomporre CFGDPQV in BCNF, abbiamo ora una terza scelta:
  - Iniziare decomponendo in FPQV e CFGDPQ
  - Poi decomporre CFGDPQ (non in 3NF) in FDP, CFGDQ
  - Questo ci dà la decomposizione in *join* senza perdita: FPQV, FDP, CFGDQ
  - Per conservare GP _ C, possiamo aggiungere CGP, come prima
- Scelta: {FPQV, FDP, CFGDQ} oppure {FDP, CFGDQV}?

Decomposizione di una relazione BCNF

- Supponiamo di scegliere {FDP, CFGDQV}. Questa è in BCNF, e non c'è ragione di decomporre ulteriormente (assumendo che tutti i VI noti siano DF)
- Però, supponiamo che queste interrogazioni siano importanti:
  - Trovare i contratti con il fornitore F
  - Trovare i contratti che coinvolgono il dipartimento D
- Decomporre ulteriormente CFGDQV in CF, CD e CGQV potrebbe velocizzare queste interrogazioni (perché?)
- D'altra parte, la seguente interrogazione è più lenta:
  - Trovare il valore totale di tutti i contratti con il fornitore F

Decomposizioni orizzontali

- La nostra definizione di decomposizione: una relazione è sostituita con una collezione di relazioni che sono *proiezioni*. Caso più importante.
- A volte, potremmo voler sostituire la relazione con una collezione di relazioni che sono *selezioni*
  - Ciascuna nuova relazione ha lo stesso schema dell'originale, ma un sottoinsieme delle righe
  - Collettivamente, le nuove relazioni contengono tutte le righe dell'originale. Tipicamente, le nuove relazioni sono disgiunte

Decomposizioni orizzontali (segue)

- Supponiamo che tutti i contratti con valore > 10.000 siano soggetti a regole diverse. Ciò significa che le interrogazioni su Contratti spesso conterranno la condizione *val* > *10000*
- Un modo di gestire ciò è la costruzione di un indice ad albero B+ *clustered* sul campo *val* di Contratti
- Un secondo approccio consiste nel sostituire i contratti con due nuove relazioni: ContrattiGrandi e ContrattiPiccoli, con gli stessi attributi (CFGDPQV)
  - Si comporta come un indice su tali interrogazioni, ma senza il sovraccarico degli indici
  - Si possono inoltre costruire indici *clustered* su altri attributi!

Mascherare i cambiamenti dello schema logico

```
CREATE VIEW Contratti(cid, fid, gid, did, pid, qta, val)
        AS SELECT *
        FROM ContrattiGrandi
        UNION
        SELECT *
        FROM ContrattiPiccoli
```

- La sostituzione di Contratti con ContrattiGrandi e ContrattiPiccoli può essere mascherata dalla vista
- Però le interrogazioni con la condizione *val* > *10000* devono essere effettuate su ContrattiGrandi per una esecuzione efficiente: quindi gli utenti interessati alle prestazioni devono essere consapevoli del cambiamento

Raffinamento delle interrogazioni e delle viste

- Se una interrogazione viene eseguita più lentamente di quanto previsto, controllare se un indice ha bisogno di essere ricostruito, o se le statistiche non siano troppo vecchie
- A volte il DBMS può non eseguire il piano che voi avete in mente. Tipiche aree di debolezza:
  - Selezioni che coinvolgono valori *null*
  - Selezioni che coinvolgono espressioni aritmetiche o stringa
  - Selezioni che coinvolgono condizioni OR
  - Mancanza di funzionalità di valutazione come strategie basate solo sull'indice o certi metodi di *join* o stime dimensionali carenti
- Controllate il piano che viene usato! Quindi aggiustate la scelta degli indici o riscrivete l'interrogazione/la vista

Riscrivere le interrogazioni SQL

- Complicate dall'interazione con
  - *NULL*, duplicati, aggregazioni, sottointerrogazioni.
- Linea guida: usare solo "blocchi di interrogazione", se possibile.

```
SELECT DISTINCT *
FROM Velisti V
WHERE V.vnome IN
        (SELECT G.vnome
                FROM GiovaniVelisti G)
```

=

```
SELECT DISTINCT V.*
FROM Velisti V,
        GiovaniVelist G
WHERE V.vnome = G.vnome
```

Non sempre possible…

```
SELECT *
FROM Velisti V
WHERE V.vnome IN
        (SELECT DISTINCT G.vnome
        FROM GiovaniVelisti G)
```

≠

```
SELECT V.*
FROM Velisti V,
        GiovaniVelisti G
WHERE V.vnome = G.vnome
```

Il noto problema del COUNT

```
SELECT dnome FROM Dipartimento D
WHERE D.num_imp >
                (SELECT COUNT(*) FROM Impiegato I
                WHERE D.palazzina = I.palazzina)
```

```
CREATE VIEW Temp (impconto, palazzina) AS
        SELECT COUNT(*), I.palazzina
        FROM Impiegato I
        GROUP BY I.palazzina
SELECT dnome
FROM Dipartimento D, Temp
WHERE D.palazzina = Temp.palazzina
AND D.num_imp > Temp.impconto;
```

- Che succede quando Impiegato è vuoto??

Riassunto sulle interrogazioni non annidate

- DISTINCT a livello superiore: si possono ignorare i duplicati
  - Si può a volte inferire DISTINCT al livello superiore! (Ad esempio la clausola della sottointerrogazione restituisce al più una tupla)
- DISTINCT nella sottointerrogazione senza DISTINCT al livello superiore: difficile da convertire
- Sottointerrogazioni con ALL: difficili da convertire
  - EXISTS e ANY sono come IN
- Aggregazioni nelle sottointerrogazioni: insidiose
- Buone notizie: alcuni sistemi ora riscrivono di nascosto (ad esempio DB2)

Altre linee guida per il raffinamento delle interrogazioni

- Minimizzare l'uso di DISTINCT: non ce n'è bisogno se i duplicati sono accettabili, o se la risposta contiene una chiave
- Minimizzare l'uso di GROUP BY e HAVING:

```
SELECT MIN(I.età)
FROM Impiegati I
GROUP BY I.rnum
HAVING I.rnum = 102
```

```
SELECT MIN(I.età)
FROM Impiegati I
WHERE I.rnum = 102
```

Consideriamo l'uso dell'indice nel DBMS quando si scrivono espressioni aritmetiche: I.età = 2 * R.età trarrà beneficio da un indice su I.età, ma potrebbe non trarne da un indice su R.età!

Altre linee guida per il raffinamento delle interrogazioni (segue)

- Evitare l'uso di relazioni intermedie

```
SELECT * INTO Temp
FROM Imp I, Rep R
WHERE I.rnum = R.rnum
        AND R.nomemgr = "Joe"
```

e

```
SELECT T.rnum, AVG(T.sal)
FROM Temp T
GROUP BY T.rnum
```

verso

```
SELECT I.rnum, AVG(I.sal)
FROM ImpI, Rep R
WHERE I.rnum = R.rnum
        AND D.nomemgr = "Joe"
GROUP BY E.rnum
```

- Non materializza la relazione intermedia Temp
- Se c'è un indice ad albero B+ denso su <rnum, sal>, si può usare un piano basato solo sull'indice per evitare di restituire le tuple di Imp nella seconda interrogazione!

Sommario

- La progettazione di una base di dati consiste di diversi passi: analisi dei requisiti, progettazione concettuale, progettazione logica, raffinamento dello schema, progettazione fisica e ottimizzazione
  - In generale, per raffinare il progetto di una base di dati si deve andare avanti e indietro su questi passi, e le decisioni prese in un passo possono influenzare le scelte da fare negli altri
- Capire la natura del carico di lavoro per l'applicazione, e gli obiettivi delle prestazioni, è essenziale per sviluppare un buon progetto
  - Quali sono le interrogazioni e gli aggiornamenti importanti? Quali attributi/relazioni sono coinvolti?

Sommario (segue)

- Lo schema logico dovrebbe essere raffinato considerando criteri relativi alle prestazioni e al carico di lavoro:
  - Si può scegliere una 3NF o una forma normale minore rispetto alla BCNF
  - Si può scegliere tra decomposizioni alternative in BCNF (o 3NF) in base al carico di lavoro
  - Si può denormalizzare, ossia annullare alcune decomposizioni
  - Si può decomporre ulteriormente una relazione BCNF!
  - Si può scegliere una decomposizione orizzontale di una relazione
  - L'importanza delle conservazione delle dipendenze è basata sulla dipendenza da conservare, e sul costo del controllo del VI
  - Si può aggiungere una relazione per garantire la conservazione delle dipendenze (per la 3NF, non per la BCNF!); oppure si può controllare la dipendenza usando un *join*

Sommario (segue)

- Nel tempo, gli indici devono essere ottimizzati (eliminati, creati, ricostruiti...) per migliorare le prestazioni
  - Si dovrebbe determinare il piano usato dal sistema, e aggiustare opportunamente la scelta degli indici
- Il sistema potrebbe ancora non trovare un buon piano:
  - Vengono considerati solo piani *left-deep*!
  - Valori *null*, condizioni aritmetiche, espressioni stringa, l'uso di OR, etc, possono confondere un ottimizzatore
- Quindi potremmo dover riscrivere l'interrogazione/la vista
  - Evitare interrogazioni annidate, relazioni temporanee, condizioni complesse, operazioni come DISTINCT e GROUP BY