BD 2 - Indici e Viste Materializzate

Luca Cosmo

Università Ca' Foscari Venezia

Università
Ca'Foscari
Venezia

Problema

Si consideri la seguente query:

```
SELECT *
FROM Movies
WHERE studio = 'Disney' AND year = 2012;
```

Se il nostro database contiene 10.000 film ma la Disney ha prodotto solo 5 film nel 2012, siamo costretti ad ispezionare 10.000 tuple per ritornare alla fine solo 5 risultati!

Intuizione

Un indice è una struttura dati ausiliaria che ci permette di accedere in maniera più efficiente alle tuple di una relazione che soddisfano una determinata proprietà.

Indici

Logicamente

Dal punto di vista logico, un indice su un attributo A di una relazione R è una lista di coppie (a_i, P_i), dove a_i è un valore di A presente in almeno una tupla di R e P_i è un insieme di puntatori alle tuple di R per cui A vale a_i. Tale lista è ordinata rispetto al valore di A.

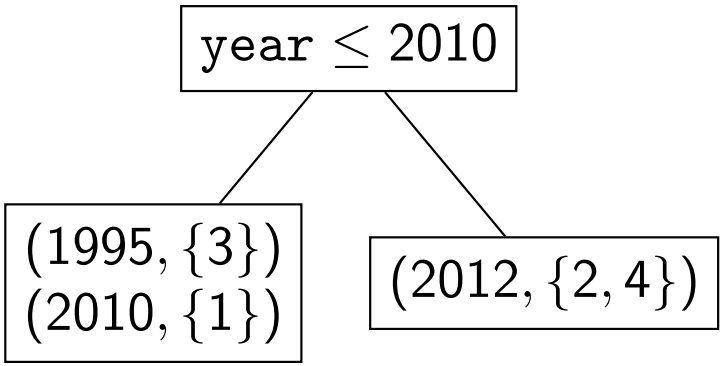
Fisicamente

Dal punto di vista fisico, un indice è spesso memorizzato in una struttura ad albero simile ad un Binary Search Tree, che consente di trovare in modo efficiente i puntatori alle tuple che soddisfano una condizione su A.

Quando un indice viene creato, esso viene utilizzato automaticamente dal query planner del DBMS quando ritenuto vantaggioso.

Indici per Attributo

Tramite un indice su `year` solo metà delle tuple devono essere esaminate per trovare i film prodotti nel 2012.

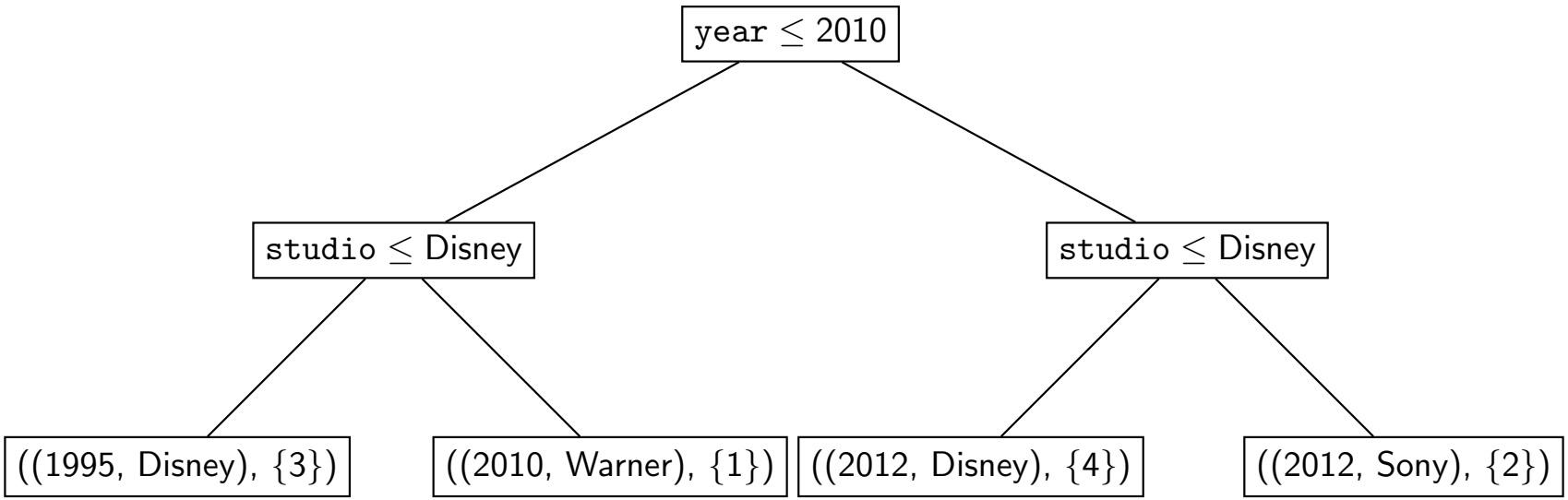


	title	year	studio
1	Inception	2010	Warner
2	Skyfall	2012	Sony
3	Toy Story	1995	Disney
4	Avengers	2012	Disney

Si noti che le tuple non sono ordinate rispetto ad `year` nella memoria fisica, ma abbiamo un modo efficiente per trovare le sole tuple rilevanti per la query.

Indici Multiattributi

E' possibile generalizzare la definizione di indice al caso di una tupla di attributi X, assumendo il classico ordinamento lessicografico fra tuple.



Indici Multiattributi

Attenzione! L'ordine degli attributi è rilevante nella costruzione di un indice multiattributo e va scelto con cura...

Indice su (`year`, `studio`)

`title`	`year`	`studio`
Toy Story	1995	Disney
Inception	2010	Warner
Avengers	2012	Disney
Skyfall	2012	Sony

Indice su (`studio`, `year`)

`title`	`year`	`studio`
Toy Story	1995	Disney
Avengers	2012	Disney
Skyfall	2012	Sony
Inception	2010	Warner

Ci aspettiamo tipicamente più query per anno o per studio?

Indici Multiattributi

Assumiamo un indice su (`year`, `studio`), quali di queste query possono trarre vantaggio dall'indice e come?

```
SELECT * FROM Movies WHERE year = 2012 AND studio = 'Sony'
```

Indici Multiattributi

Assumiamo un indice su (`year`, `studio`), quali di queste query possono trarre vantaggio dall'indice e come?

```
SELECT * FROM Movies WHERE year = 2012 AND studio = 'Sony'
```

```
SELECT * FROM Movies WHERE studio = 'Sony'
```

Indici Multiattributi

Assumiamo un indice su (`year`, `studio`), quali di queste query possono trarre vantaggio dall'indice e come?

```
SELECT * FROM Movies WHERE year = 2012 AND studio = 'Sony'
```

```
SELECT * FROM Movies WHERE studio = 'Sony'
```

```
SELECT * FROM Movies WHERE year = 2012
```

Indici Multiattributi

Assumiamo un indice su (`year`, `studio`), quali di queste query possono trarre vantaggio dall'indice e come?

```
SELECT * FROM Movies WHERE year = 2012 AND studio = 'Sony'
```

```
SELECT * FROM Movies WHERE studio = 'Sony'
```

```
SELECT * FROM Movies WHERE year = 2012
```

```
SELECT * FROM Movies WHERE year > 1998 AND year < 2012
```

Indici Multiattributi

Assumiamo un indice su (year, studio), quali di queste query possono trarre vantaggio dall'indice e come?

```
SELECT * FROM Movies WHERE year = 2012 AND studio = 'Sony'
```

```
SELECT * FROM Movies WHERE studio = 'Sony'
```

```
SELECT * FROM Movies WHERE year = 2012
```

```
SELECT * FROM Movies WHERE year > 1998 AND year < 2012
```

```
SELECT * FROM Movies WHERE year = 2012 ORDER BY studio
```

Esempio 1

Assumiamo che ci siano 10.000 film e che la Disney ne abbia prodotti 200, di cui solo 5 nel 2012.

```
SELECT *
FROM Movies
WHERE studio = 'Disney'
      AND year = 2012;
```

Scelte possibili:

- nessun indice: 10.000 tuple
- indice su `studio`: 200 tuple
- indice multiattributo su `(studio, year)`: 5 tuple

Si noti che l'uso di indici non aiuta solo le ricerche su una singola tabella, ma è molto utile anche nel caso di join.

Esempio 2

Assumiamo che ci siano 10.000 film e 500 produttori.

```
SELECT e.name
FROM Movies m, MovieExec e
WHERE m.title = 'Star Wars'
    AND m.producer = e.code;
```

Scelte possibili:

- nessun indice: 10.500 tuple
- indice su `m.title`: 501 tuple
- indice su `m.title` ed indice su `e.code`: 2 tuple

Ma quanto sono importanti questi numeri in pratica? Se l'intero database fosse in RAM probabilmente poco, ma questo non è realistico!

Tuple o Pagine?

Il numero di tuple è una misura imprecisa del costo di un'operazione, perchè ignora in larga parte l'organizzazione fisica della memoria.

Una metrica migliore è basata sul numero di pagine caricate in RAM:

- ciascuna pagina tipicamente contiene molte tuple
- anche una singola tupla richiede che l'intera pagina corrispondente sia caricata in RAM
- l'accesso a tutte le tuple in una pagina è solo leggermente più costoso dell'accesso ad una singola tupla

Il numero di pagine accedute è spesso funzione del numero di tuple accedute... anche se bisogna fare attenzione!

Tuple o Pagine?

Se una tabella è fortemente "clusterizzata" su un certo attributo nella memoria fisica, è possibile accedere a molte tuple caricando solo poche pagine: il numero di tuple è una stima pessimistica del costo effettivo.

```
SELECT *
FROM Movies
WHERE year = 2001;
```

Supponiamo che la tabella `Movies` occupi 700 pagine, che ogni pagina contenga 100 tuple e che vi siano 300 film prodotti nel 2001.

Quante pagine devono essere caricate in RAM con un indice su `year`?

- se i film sono salvati su disco per anno, possono bastare 3 pagine
- ma nel caso peggiore potrebbero servire addirittura 300 pagine!

Indici: Pro e Contro

Pro

Un indice su un attributo può accelerare di molto l'esecuzione delle query in cui un valore (o un intervallo di valori) è specificato per quell'attributo, oltre che le join che coinvolgono quell'attributo.

Contro

Ciascun indice costruito su una tabella rende le operazioni di inserimento, cancellazione ed aggiornamento su quella tabella più costose, dato che anche l'indice deve essere aggiornato.

Suggerimenti

Quando definire un indice?

- Su una chiave (o una "quasi" chiave)
- Sulle chiavi esterne (per semplificare le join)
- Quando le operazioni di modifica sono rare
- Quando le tuple sono "clusterizzate" su un certo attributo nella memoria fisica (vedi comando `CLUSTER` di Postgres)

Quando NON definire un indice?

- Su tabelle piccole, che quindi occupano un numero ridotto di pagine
- Su attributi poco selettivi (es. sesso o stato civile)
- Su attributi modificati di frequente

Definizione di Indici

Una sintassi tipica per definire un nuovo indice è la seguente:

```
CREATE INDEX NomeIndice ON NomeTabella (Attributi);
```

Una sintassi tipica per l'eliminazione di un indice è la seguente:

```
DROP INDEX NomeIndice;
```

Una volta che un indice è definito, il DBMS lo usa automaticamente:

- usate `EXPLAIN` per verificare che gli indici siano usati quando ve lo aspettereste: potreste avere sbagliato a definirli o la distribuzione dei dati potrebbe essere cambiata senza che il query planner lo sappia

EXPLAIN: Esempio

```
EXPLAIN SELECT * FROM foo;

                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on foo  (cost=0.00..155.00 rows=10000 width=4)
(1 row)
```

```
EXPLAIN SELECT * FROM foo WHERE i = 4;

                          QUERY PLAN
--------------------------------------------------------------
 Index Scan using fi on foo  (cost=0.00..5.98 rows=1 width=4)
   Index Cond: (i = 4)
(2 rows)
```

Modello di Costo

Consideriamo la relazione `StarsIn(movie, year, starName)` ed i seguenti pattern di operazioni tipiche.

Q_1

```
SELECT movie, year
FROM StarsIn
WHERE starName = s
```

Q_2

```
SELECT starName
FROM StarsIn
WHERE movie = t
      AND year = y
```

I

```
INSERT
INTO StarsIn
VALUES(m, y, s)
```

Costo di Q_1

```
SELECT movie, year
FROM StarsIn
WHERE starName = s
```

Assumendo che `StarsIn` occupi 10 pagine e che in media ogni attore abbia recitato in 3 film, abbiamo i seguenti costi:

- senza indice: 10
- indice su `starName`: $1 + 3$
- indice su `(movie, year)`: 10
- entrambi gli indici: $1 + 3$

Costo di Q_2

```
SELECT starName
FROM StarsIn
WHERE movie = t
      AND year = y
```

Assumendo che `StarsIn` occupi 10 pagine e che in media ogni film abbia 3 attori, abbiamo i seguenti costi:

- senza indice: 10
- indice su `starName`: 10
- indice su `(movie, year)`: $1 + 3$
- entrambi gli indici: $1 + 3$

Costo di *I*

```
INSERT
INTO StarsIn
VALUES(m, y, s)
```

Assumendo che `StarsIn` occupi 10 pagine e che in media ogni film abbia 3 attori, abbiamo i seguenti costi:

- senza indice: 2
- indice su `starName`: $2 + 2$
- indice su `(movie, year)`: $2 + 2$
- entrambi gli indici: $2 + 2 + 2$

Cosa Fare?

Supponiamo di eseguire Q_1 con probabilità p_1, Q_2 con probabilità p_2 ed I con probabilità $1 - p_1 - p_2$.

Operazione	Nessuno	Primo	Secondo	Entrambi
Q_1	10	4	10	4
Q_2	10	10	4	4
I	2	4	4	6
Costo	$2 + 8p_1 + 8p_2$	$4 + 6p_2$	$4 + 6p_1$	$6 - 2p_1 - 2p_2$

Se $p_1 = 0.4$ e $p_2 = 0.1$, otteniamo che la soluzione migliore è usare solo il primo indice (su `starName`).

Selezione Automatica di Indici

Un DBMS può suggerire automaticamente gli indici migliori sulla base di un modello di costo simile a quello considerato:

1. usa i log delle query per stimare il costo delle operazioni più frequenti
2. genera un insieme di indici candidati I e stima i tempi di esecuzione rispetto ai vari $J \subseteq I$
3. ritorna l'insieme di indici J_{min} che ottimizza i tempi di esecuzione

Il secondo passo potrebbe essere implementato in maniera greedy per motivi di efficienza.

Viste e Performance

Nel primo modulo del corso avete studiato l'uso di viste:

```
CREATE VIEW MovieProd AS
        SELECT m.title, m.year, e.name
        FROM Movies m, MovieExec e
        WHERE m.producer = e.code
```

Questa vista è utile quando vogliamo trovare il nome del produttore di un film, ma deve essere valutata ogni volta che ci facciamo una query sopra. Questo è potenzialmente inefficiente.

Viste Materializzate

SQL permette di materializzare una vista in memoria, in modo che essa non venga valutata ad ogni query che la coinvolge:

```
CREATE MATERIALIZED VIEW MovieProd AS
        SELECT m.title, m.year, e.name
        FROM Movies m, MovieExec e
        WHERE m.producer = e.code
```

L'uso di viste materializzate può migliorare le prestazioni delle query, ma comporta un costo aggiuntivo derivante dalla necessità di riflettere sulla vista le modifiche alle tabelle su cui la vista è costruita.

Viste Materializzate: Mantenimento

```
CREATE MATERIALIZED VIEW MovieProd AS
        SELECT m.title, m.year, e.name
        FROM Movies m, MovieExec e
        WHERE m.producer = e.code
```

Non c'è bisogno di aggiornare `MovieProd` nei seguenti casi:

- modifiche a tabelle diverse da `Movies` e `MovieExec`
- modifiche ad attributi di `Movies` e `MovieExec` diversi da quelli menzionati nella definizione di `MovieProd`

Approccio conservativo: in tutti gli altri casi rigeneriamo la vista, ma ci sono diverse ottimizzazioni. Postgres delega la responsabilità all'utente tramite il comando `REFRESH MATERIALIZED VIEW`.

Viste Materializzate: Ottimizzazioni

```
CREATE MATERIALIZED VIEW MovieProd AS
        SELECT m.title, m.year, e.name
        FROM Movies m, MovieExec e
        WHERE m.producer = e.code
```

Assumiamo di voler aggiungere un nuovo film: Kill Bill, prodotto nel 2003 dal produttore con codice 23456 (Quentin Tarantino).

In questo caso non serve rigenerare `MovieProd`:

```
INSERT INTO MovieProd
VALUES ('Kill Bill', 2003, 'Quentin Tarantino');
```

Viste Materializzate: Ottimizzazioni

```
CREATE MATERIALIZED VIEW MovieProd AS
        SELECT m.title, m.year, e.name
        FROM Movies m, MovieExec e
        WHERE m.producer = e.code
```

Assumiamo di voler cancellare un film: Il Re Leone, prodotto nel 1994.

In questo caso non serve rigenerare `MovieProd`:

```
DELETE FROM MovieProd
WHERE title = 'Il Re Leone' AND year = 1994;
```

Viste Materializzate e Performance

In sostanza, le viste materializzate:

- possono velocizzare le query, ma rendono le operazioni di modifica più costose (come gli indici) o potenzialmente invalidanti (Postgres)
- in certi DBMS vengono mantenute in modo incrementale per quanto possibile, ma richiedono di essere rigenerate dopo certe operazioni
- possono essere inlined automaticamente dal DBMS in una query per migliorarne l'efficienza, sfruttando il fatto che parte dell'informazione è già stata computata e salvata in memoria

Inlining di Viste Materializzate

Vista materializzata V

```
SELECT AV
FROM RV
WHERE CV
```

Query Q

```
SELECT AQ
FROM RQ
WHERE CQ
```

Condizioni per l'inlining di V in Q:

1. RV $\subseteq$ RQ
2. CQ = CV AND CQ' per qualche CQ'
3. ogni attributo di CQ' che proviene da RV fa parte di AV
4. ogni attributo di AQ che proviene da RV fa parte di AV

Risultato dell'inlining:

```
SELECT AQ
FROM V, RQ \ RV
WHERE CQ'
```

Inlining di Viste Materializzate: Esempio

Vista materializzata V

```
CREATE MATERIALIZED VIEW MovieProd AS
        SELECT m.title, m.year, e.name
        FROM Movies m, MovieExec e
        WHERE m.producer = e.code
```

Query Q

```
SELECT s.starName
FROM StarsIn s, Movies m, MovieExec e
WHERE s.title = m.title
      AND s.year = m.year
      AND m.producer = e.code
      AND e.name = 'Tarantino'
```

Risultato dell'inlining

```
SELECT s.starName
FROM StarsIn s, MovieProd mp
WHERE s.title = mp.title
      AND s.year = mp.year
      AND mp.name = 'Tarantino'
```

Checkpoint

Punti Chiave

- Definizione di indici, con i loro pro e contro
- Il modello di costo per identificare la scelta degli indici migliori
- Definizione di viste materializzate, con i loro pro e contro

Materiale Didattico

Database Systems: Sezioni 8.3, 8.4 e 8.5. Approfondimenti dal manuale di Postgres (indexes e materialized views).