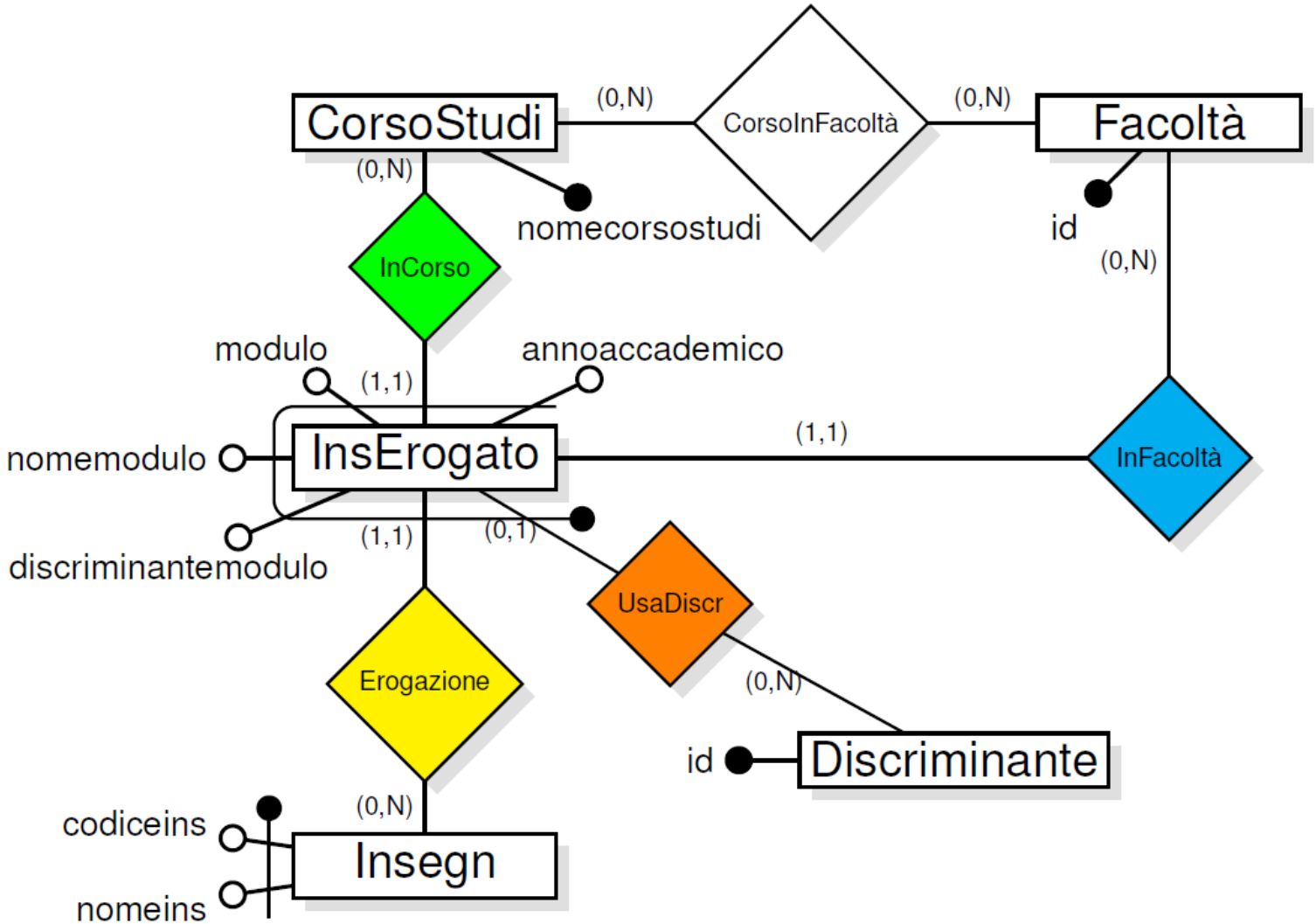
Basi di Dati
Modulo Laboratorio

Lezione 7: Introduzione agli indici e analisi prestazioni query SQL

DR. SARA MIGLIORINI

Indici

- Gli indici sono delle strutture dati ausiliare che permettono di accedere ai dati di una tabella in maniera più efficiente.
- Dato che un indice è una struttura dati ausiliaria, deve essere sempre mantenuto aggiornato in base al contenuto della tabella in cui è definito.
- Il costo di aggiornamento può essere significativo quando ci sono molti indici definiti sulla medesima tabella.
- Gli indici quindi devono essere definiti considerando la loro efficacia.
- In questa lezione si introducono gli indici e come valutare la loro efficacia.



Si rappresentano solo le chiavi.

Base di dati UNIVR – Schema concettuale

Base di dati UNIVR – Schema relazionale

- Scelte progettuali:

1. Ogni tabella ha l'attributo id che è chiave primaria (scelta imposta dal tool di progettazione). La sottolineatura ondulata indica gli attributi che costituiscono la chiave concettuale.
2. In ogni tabella, una chiave esportata ha formato id_<tabellaEsterna> e si riferisce sempre all'attributo id della tabella esterna.

Base di dati UNIVR – Schema relazionale

Insegn		
id	nomeins	codiceins

Discriminante		
id	nome	descrizione

InsErogato					
id	id_isegn	id_discriminante	annoaccademico	modulo	crediti
id_corsostudi	nomemodulo	discriminantemodulo	programma	id_facoltà	…

CorsoStudi			
id	nomecorsostudi	sede	durataAnni

CorsoInFacolta		
id	id_corsostudi	id_facoltà

Facoltà	
id	nome

Base di dati UNIVR – Schema fisico

- InsErogato(id, id_insegn, id_corsostudi, id_discriminante, annoaccademico, id_facolta, modulo, nomemodulo, discriminantemodulo, crediti, programma)
- Insegn(id, nomeins, codiceins)
- Discriminante(id, nome, descrizione)
- CorsoStudi(id, nomecorsostudi, sede, durataAnni)
- CorsoInFacolta(id, id_corsostudi, id_facolta)
- Facolta(id, nome)

Indici – Caso pratico

- La base di dati did2014small contiene le tabelle Insegn, Discriminante, InsErogato e CorsoStudi della base di dati did2014 senza alcuna chiave vincolo referenziale e indice.
- Si useranno queste tabelle per studiare gli effetti degli indici.
- Caricare le tabelle nella propria base di dati personale facendo un backup di did2014small e poi un restore nella propria base di dati:

Indici – Caso pratico

In una shell dare i seguenti comandi

$ pg_dump -xcO -h dbserver.scienze.univr.it -d did2014small -U <login GIA> -f backupDid2014small .sql
... #Il file backupDid2014small .sql contiene il backup in formato SQL !

$ psql -h dbserver . scienze . univr .it -d <login GIA> -U <login GIA> -f backupDid2014small .sql
SET ...
psql : backupDid2014small .sql :23: ERRORE : ...
...
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
COPY 635
COPY 136
COPY 8169
COPY 68017
Import può dare degli errori di autorizzazione: sono da ignorare.

Indici – Caso pratico: Query SENZA Indici

- Si consideri la query:

```
SELECT id, nomeins FROM Insegn WHERE nomeins='Algoritmi';
```

- Il DBMS deve fare una scansione sequenziale della tabella Insegn rispetto nomeins per estrarre le righe con valore uguale a 'Algoritmi';
- Il comando \timing attiva/disattiva la visualizzazione del tempo di pianificazione + calcolo di una query.
- Il tempo è visualizzato subito dopo la visualizzazione del risultato della query.
- \timing dà un'idea del tempo necessario per eseguire una query.
- Prima di usare psql, impostare la variabile d'ambiente LANG: export LANG='en_US.UTF-8';

Indici – Caso pratico: Query SENZA Indici

Query SENZA Indici

```
=>\ timing
Timing is ON.
=> SELECT id , nomeins
FROM Insegn
WHERE nomeins ='Algoritmi ‘;

id    | nomeins
------+--------------
5441 | Algoritmi

(1 row)
TIME : 2.437 ms
```

- PostgreSQL ha risposto in inglese: ‘.’ separa i decimali.
- Il tempo è di 2 millisecondi circa.

Indici – Sintassi

Sintassi CREATE INDEX semplificata

```
CREATE INDEX [ nome ]
ON nomeTabella [ USING method ]
({ nomeAttr |( expression )} [ASC| DESC ] [, ...])
```

- method è il tipo di indice (più avanti i dettagli);
- nomeAttr o expression indicano su quali espressione di attributi si deve creare indice.
- ASC/DESC specifica se l'indice è ordinato in modo ascendente o discendente.
- ALTER INDEX e DROP INDEX permettono di modificare/rimuovere indici creati precedentemente.

Indici – Caso pratico: Query CON Indici

- Per rendere più veloce l'esecuzione della query
- SELECT id, nomeins FROM Insegn WHERE nomeins='Algoritmi';
- un indice sull'attributo nomeins potrebbe essere utile perché nomeins è l'attributo usato per selezionare le righe.

Query CON indici

```
=> CREATE INDEX nomeins_index ON Insegn ( nomeins );
=> ANALYZE Insegn;
=> SELECT id, nomeins FROM Insegn WHERE nomeins ='Algoritmi’;

id   | nomeins
-----+---------------
5441 | Algoritmi
TIME : 0.587 ms
```

Indici – Caso pratico

- Il tempo di esecuzione è passato da 2.4337 ms a 0.587 ms.
- Nota: Non è garantito che il tempo di esecuzione sia sempre uguale perché il server è multitask. I tempi di queste slide sono stati determinati con il server completamente dedicato.

Indici

- Un indice, una volta creato, è usato dal sistema ogni volta che l'ottimizzatore di query determina sia opportuno.
- Un DBMS crea gli indici in modo automatico solo per attributi dichiarati PRIMARY KEY.
- Per tutti gli altri attributi si deve fare una dichiarazione esplicita di CREATE INDEX per attivarlo.
- Un indice può velocizzare anche i comandi UPDATE/DELETE quando nella clausola WHERE ci sono attributi indicizzati.
- Il comando ANALYZE [nomeTabella] aggiorna le statistiche circa il contenuto delle tabelle (e indici). È eseguito in modo automatico dal sistema a intervalli regolari.
- L'ottimizzatore di query usa queste statistiche per decidere quando usare gli indici.

Indici – Caso pratico

Query SENZA Indici

Si consideri la query

```
SELECT DISTINCT I.nomeins
FROM CorsoStudi CS
  JOIN InsErogato IE ON CS.id = IE.id_corsostudi
  JOIN Insegn I ON I.id =IE.id_insegn
WHERE IE.annoaccademico = '2009/2010'
  AND CS.nome = 'Laurea in Informatica';
...
(26 ROWS )
TIME : 40.975 ms
```

Indici – Caso pratico

- Quanto è possibile rendere più veloce la query definendo degli indici?
- Si ricorda: gli indici sono usati per indirizzare tuple che contengono i valori dati sugli attributi specificati.
- Una possibilità quindi è definire gli indici su tutti gli attributi usati per fare i JOIN o le selezioni.

Indici – Caso pratico

Creazione indici

Si considerino quindi gli indici:

```
CREATE INDEX cs_id ON CorsoStudi(id);
CREATE INDEX i_id ON Insegn(id);
```

- NOTA: gli indici sugli attributi PRIMARY KEY (id nel nostro caso) solitamente sono già presenti!
  - In did2014small sono stati tolti per fare gli esperimenti!

- continua…

Indici – Caso pratico

Creazione indici

```
CREATE INDEX ie_id_corsostudi ON Inserogato( id_corsostudi );
CREATE INDEX ie_id_insegn ON Inserogato( id_insegn );
CREATE INDEX ie_aa ON Inserogato( annoaccademico );
CREATE INDEX cs_nome ON Corsostudi( nome );
ANALYZE Corsostudi;
ANALYZE Inserogato;
ANALYZE Insegn;
```

- Non tutti gli indici sono necessari, ma per ora accettiamo il costo di crearli tutti.
- In seguito si presenterà uno strumento per verificare quali indici tenere e quali cancellare.
- Importante! Forziamo il DBMS ad aggiornare le statistiche con il comando ANALYZE dopo la creazione degli indici.

Indici – Caso pratico

Query CON Indici

Ora la query richiede un tempo inferiore:

```
SELECT DISTINCT I.nomeins
FROM CorsoStudi CS
   JOIN InsErogato IE ON CS.id = IE.id_corsostudi
   JOIN Insegn I ON I.id = IE.id_insegn
WHERE IE.annoaccademico = '2009/2010'
   AND CS. nome = 'Laurea in Informatica';
...
(26 ROWS )
TIME : 5.201 ms
```

Il tempo di esecuzione è passato da ~40 ms a ~5 ms.

Indici – Tipi: B-tree

- PostgreSQL ammette diversi tipi di indici: B-tree, hash, GiST, SP-GiST, GIN e BRIN.
- Ciascun tipo usa una tecnica algoritmica diversa e risulta migliore di altri per specifici tipi di query.
- Sintassi per specificare anche il tipo:

```
CREATE INDEX nome ON nomeTab USING type(COLUMN);
```

- Se non si specifica il tipo di indice nel CREATE INDEX, il sistema crea un indice di tipo B-tree.
- L'ottimizzatore di query considera un indice B-tree ogni volta che l'attributo indicizzato è coinvolto in un confronto usando uno degli operatori: <, <=, =, >=, >.
- Un indice B-tree può essere considerato anche quando ci sono BETWEEN, IN, IS NULL, IS NOT NULL e LIKE.

Indici – Tipi [NOTA]

- Con la localizzazione (it_IT.UTF-8), la comparazione di stringhe con operatori diversi da '=' e con LIKE ha regole diverse (minuscole/maiuscole) rispetto a UTF8.
- Un indice su attributo VARCHAR in una base di dati con LC_LOCALE = it_IT.UTF-8 (come le basi di dati in dbserver) dovrebbe essere dichiarato come:

  CREATE INDEX nome ON nomeTab(nomeAttr varchar_pattern_ops);

- se si vuole che l'indice sia usato anche quando il confronto usa <,>,<>, LIKE.
- La parola chiave varchar_pattern_ops abilita l'uso degli operatori rispettando le regole imposte dal locale (it_IT.UTF-8).

Indici – Tipi: Hash

- Il tipo hash ha un uso limitato perché può essere usato solo quando i confronti sono di eguaglianza.
- La sua gestione poi è più complicata in basi di dati replicate o in caso di crash: il gruppo di PostgreSQL ne sconsiglia l'uso.
- Gli altri tipi di indici sono indicati per particolari strutture dati. Per esempio, GiST è indicato quando un attributo è di tipo geometrico bi-dimensionale (tipo non standard).
- Vedere capitolo 11.2 manuale di PostgreSQL per maggiori dettagli.

Indici – Multi-attributo

- Se si hanno query che hanno condizioni su coppie (terne, ecc) di attributi di una tabella, un indice multi-attributo definito usando la coppia (terna, ecc) di attributi potrebbe essere più utile rispetto gli indici sui singoli attributi.

Query con indici semplici

```
SELECT I.nomeins , I.codiceins
FROM Insegn I
    JOIN InsErogato IE ON I.id = IE.id_insegn
WHERE IE.annoaccademico = '2006/2007'
    AND IE.id_corsostudi = 4;
...(46 ROWS )

TIME : 2.884 ms
```

Già ottimizzata usando gli indici precedenti: ie_id_corsostudi e ie_aa.

Indici – Multi-attributo

- Clausola WHERE seleziona le righe confrontando con costanti due attributi della tabella Inserogato: si può definire un indice multi-attributo.

Creazione indice multi-attributo

```
CREATE INDEX ie_aa_idcs ON Inserogato( annoaccademico, id_corsostudi );
ANALYZE Inserogato;
```

Query con indice multi-attributo

```
SELECT I.nomeins, I.codiceins
FROM Insegn I JOIN InsErogato IE ON I.id = IE. id_insegn
WHERE IE. annoaccademico = '2006/2007 '
AND IE. id_corsostudi = 4;
...(46 ROWS )
TIME : 1.910 ms
```

Indici – Multi-attributo

NOTA

NON sempre gli indici multi-attributo possono essere usati: per esempio in espressioni con OR non è possibile.

```
SELECT I. nomeins , I. codiceins
FROM Insegn I JOIN InsErogato IE ON I.id = IE. id_insegn
WHERE IE. annoaccademico = '2006/2007' OR IE. id_corsostudi = 4;
...(5334 ROWS )
```

- Non c'è il tempo di esecuzione perché il risultato è diverso e quindi non comparabile.
- In questa query l'indice ie_aa_idcs NON è usato perché la disgiunzione OR rende impossibile di fatto il suo uso.
- Nemmeno gli indici singoli ie_id_corsostudi e ie_aa sono usati.
- L'ottimizzatore esegue una scansione di tutta la tabella Insegn.

Indici di espressioni

- Query con condizioni su espressioni/funzioni di uno o più attributi di una tabella possono essere velocizzate creando indici su le medesime espressioni/funzioni di attributi.

Query con espressione che non usa un indice semplice

```
CREATE INDEX ins_nomeins ON Insegn( nomeins varchar_pattern_ops );
ANALYZE Insegn;

SELECT nomeins FROM Insegn
WHERE nomeins LIKE 'Algoritmi %';
...(4 ROWS )
TIME : 1.718 ms

SELECT nomeins FROM Insegn
WHERE LOWER ( nomeins ) LIKE 'algoritmi %';
...(4 ROWS )
TIME : 8.696 ms
```

Per la seconda query non si può usare l'indice ins_nomeins perché nella clausola WHERE c'è LOWER(nomeins).

Indici di espressioni

- È possibile definire indici anche usando espressioni di attributi. Sintassi:

CREATE INDEX nome ON nomeTabella(expression);

- dove expression è una espressione su uno o più attributi.
- Per opportunità, si considerano espressioni che sono frequenti nelle interrogazioni usate.

Query con indici definiti usando espressioni

```
CREATE INDEX ins_lower_nonins ON Insegn ( LOWER ( nomeins )
varchar_pattern_ops );
ANALYZE Insegn;
SELECT nomeins FROM Insegn
WHERE LOWER ( nomeins ) LIKE 'algoritmi %';
...(4 ROWS )
TIME : 1.796 ms
```

Indici – Costo e regola pratica d'uso

- Gli indici costano tempo e memoria.
- Se si considera solo il tempo, il costo maggiore è mantenere gli indici aggiornati.
- Per ogni operazione di INSERT, UPDATE, DELETE su una tabella con indici, il DBMS deve aggiornare anche gli indici della tabella.
- Regola pratica: definire gli indici in base alle query più frequenti [cap.11.11].
- Il comando PostgreSQL \di visualizza l'elenco degli indici definiti in una base di dati.
- Il comando PostgreSQL \d nomeTabella visualizza la definizione di tabella e degli indici associati.

Explain

- Un DBMS ha un ottimizzatore di query che determina un piano per eseguire una data query nel minor tempo possibile.
- Il comando EXPLAIN query [cap. 14.1] permette di vedere il piano di esecuzione di una query che l'ottimizzatore determina senza
- eseguire la query.
- La corretta interpretazione dell'output di un EXPLAIN richiede una certa esperienza e conoscenza dei meccanismi dell'ottimizzatore.
- In questa lezione si introducono gli aspetti fondamentali del comando EXPLAIN, sufficienti per individuare le cause più comuni di query con basse prestazioni.
- Un piano di esecuzione di una query è un albero di nodi di esecuzione.
- Le foglie sono nodi di scansione: l'esecuzione di questi nodi restituiscono indirizzi di righe della tabella.

Explain - Struttura

- Esistono differenti tipi di scansioni: sequenziali, indicizzate e mappate su bit.
- Se una query contiene JOIN, GROUP BY o ORDER BY o altre operazioni sulle righe, allora ci saranno altri nodi di esecuzione sopra le foglie nell'albero.
- L'output di EXPLAIN ha una riga per ciascun nodo nell'albero di esecuzione dove viene indicato il tipo di operazione e una stima del costo di esecuzione.
- Ulteriori proprietà del nodo possono essere mostrate con una riga indentata subito sotto.
- La prima riga dell'output ha la stima del costo totale di esecuzione della query. Questo è il costo che l'ottimizzatore cerca di minimizzare.

Explain - Struttura

Esempio di EXPLAIN di una sola riga

```
EXPLAIN SELECT * FROM Insegn;

QUERY PLAN
-----------------------------------------------------------
Seq Scan ON insegn (cost=0.0..185.69 ROWS=8169 width=63)
```

- 0.0 è il costo iniziale, per produrre la prima riga.
- 185.69 è il costo totale, per produrre tutte le righe.
- 8169 è il numero totale di righe del risultato
- 63 è la dimensione, in byte, di ciascuna riga.

NOTA

- Il costo è in termini di numero di accessi alla memoria secondaria.
- Il numero totale di righe non è sempre il numero totale di righe valutate dall'esecutore.

Explain - Struttura

Esempio di EXPLAIN con due nodi

```
EXPLAIN SELECT * FROM Insegn WHERE id < 1000;

QUERY PLAN
-----------------------------------------------------------
Bitmap Heap Scan ON insegn (cost=18.60..132.79 ROWS=815...)
Recheck Cond: (id < 1000)
-> Bitmap INDEX Scan ON i1 (cost=0..18.39 ROWS=815 w =0)
   INDEX Cond: (id < 1000)
```

- Prima viene eseguito il nodo foglia (Bitmap INDEX Scan): grazie all'indice B-tree, l'ottimizzatore determina un vettore di indirizzi di righe da considerare (in un B-tree se la chiave cercata k non è pari a nessuna chiave K_i ma è compresa tra K_i e K_{i+1}, può essere presente nel sottoalbero P_i).
- Tale vettore viene poi passato al nodo padre (Bitmap Heap Scan) che carica le righe ed esegue la selezione finale ricontrollando che id < 1000.

Explain - Struttura

Esempio di EXPLAIN con due nodi

```
DROP INDEX ins_nomeins; -- c'è solo indice su id
EXPLAIN SELECT * FROM Insegn WHERE id <1000 AND nomeins LIKE 'Al%';

QUERY PLAN
-----------------------------------------------------------
Bitmap Heap Scan ON Insegn (cost=18.40..134.62 ROWS=1...)
Recheck Cond : (id < 1000)
Filter: (( nomeins )::TEXT ~~ 'Al%'::TEXT )
-> Bitmap INDEX Scan ON i1 (cost=0.00..18.39 ROWS=815 width =0)
   INDEX Cond: (id < 1000)
```

- Il nodo Bitmap Heap Scan carica le righe indicate dall'indice e, per ciascuna, ricontrolla che id < 1000, e la seleziona se nomeins inizia con 'Al'.

Explain - Struttura

Esempio di EXPLAIN con uso di due indici sulla stessa tabella

```
CREATE INDEX nomeIdx ON Insegn (nomeins varchar_pattern_ops);
ANALYZE Insegn;
EXPLAIN SELECT * FROM Insegn WHERE id <1000 AND nomeins LIKE 'Al%';

QUERY PLAN
-----------------------------------------------------------
Bitmap Heap Scan ON insegn (cost=15.30..22.58 ROWS=4 width=63)
Recheck Cond: (id < 1000)
Filter: (( nomeins ):: TEXT ~~ 'Al%':: TEXT )
-> BitmapAnd (cost=15.30..15.30 ROWS=4 width=0)
-> Bitmap INDEX Scan ON nomeidx (cost=0..2.65 ROWS=37 width=0)
   INDEX Cond: (((nomeins)::TEXT ~>=~ 'Al'::TEXT ) AND
                ((nomeins)::TEXT ~<~ 'Am '::TEXT ))
-> Bitmap INDEX Scan ON i1 (cost=0.00..12.40 ROWS=815 width=0)
   INDEX Cond: (id < 1000)
```

- Ciascuna foglia scansiona un indice. Il padre delle foglie (BitmapAnd) fa l'intersezione degli indirizzi delle righe trovate.
- La root (Bitmap Heap Scan) carica le righe ed ricontrolla che le condizioni siano soddisfatte.

Explain - Struttura

Esempio di EXPLAIN con uso di due indici sulla stessa tabella

```
EXPLAIN SELECT * FROM Insegn WHERE id <1000 OR nomeins LIKE 'Al%';

QUERY PLAN
-----------------------------------------------------------
Bitmap Heap Scan ON insegn (cost=15.47..132.25 ROWS=850 width =63)
Recheck Cond: ((id < 1000) OR ((nomeins)::TEXT ~~ 'Al%'::TEXT ))
Filter : ((id < 1000) OR ((nomeins)::TEXT ~~ 'Al%'::TEXT ))
-> BitmapOr (cost=15.47..15.47 ROWS=852 w=0)
-> Bitmap INDEX Scan ON i1 (cost=0.00..12.40 ROWS=815 width=0)
   INDEX Cond: (id < 1000)
-> Bitmap INDEX Scan ON nomeidx (cost=0.00..2.65 ROWS=37 width=0)
   INDEX Cond: (((nomeins)::TEXT ~>=~ 'Al'::TEXT ) AND
                ((nomeins)::TEXT ~<~ 'Am':: TEXT ))
```

- Il padre delle foglie (BitmapOr) esegue l'unione degli indirizzi delle righe trovate.

Explain - Struttura

Esempio di EXPLAIN con planner che decide diversamente cambiando condizione

```
EXPLAIN SELECT * FROM Insegn WHERE id <1000 AND nomeins LIKE '%Al';

QUERY PLAN
-----------------------------------------------------
Bitmap Heap Scan ON insegn (cost=12.40..128.62 ROWS=1 width=63)
Recheck Cond: (id < 1000)
Filter: ((nomeins)::TEXT ~~ '%Al'::TEXT )
-> Bitmap INDEX Scan ON i1 (cost=0.00..12.40 ROWS=815 width=0)
   INDEX Cond: (id < 1000)
```

- La condizione nomeins LIKE '%Al' rende impossibile l'uso dell'indice.

Explain - Struttura

Esempio di EXPLAIN di un JOIN: join mediante hash table

```
EXPLAIN SELECT * FROM Insegn I JOIN Inserogato IE ON ie. id_insegn = i.id
WHERE ie.annoaccademico = '2013/2014';

QUERY PLAN
-----------------------------------------------------------
Hash JOIN (cost=287.80..6328.90 ROWS=5155 width=641)
Hash Cond: (ie. id_insegn = i.id)
-> Seq Scan ON inserogato ie (cost =0.00..5970.21 ROWS =5155 w =578)
   Filter : ((annoaccademico)::TEXT = '2013/2014'::TEXT )
-> Hash (cost =185.69..185.69 ROWS =8169 width =63)
-> Seq Scan ON insegn i (cost =0.00..185.69 ROWS =8169 w =63)
```

- Rimossi indici relativi ad annoaccademico prima di eseguire questa prova.
- Il nodo Hash prepara un hash table tramite una scansione sequenziale Seq Scan.
- Il nodo Hash Join, per ogni riga fornita dal primo figlio, Seq Scan, cerca nella hash table (secondo figlio) la riga da unire secondo la condizione.
- Nonostante Insegn.id ha indice, qui non viene usato!
- L'unico Seq Scan che si potrebbe ottimizzare è quello evidenziato.

Explain - Struttura

Esempio di EXPLAIN di un JOIN: join mediante nested loop

```
EXPLAIN SELECT * FROM Insegn I JOIN Inserogato IE ON ie.id_insegn > i.id
WHERE ie. annoaccademico = '2013/2014';

QUERY PLAN
-----------------------------------------------------------
Nested Loop (cost=0.28..393703.79 ROWS=14037065 width=641)
-> Seq Scan ON inserogato ie (cost=0.00..5970.21 ROWS=5155 w=578)
   Filter: (( annoaccademico )::TEXT = '2013/2014':: TEXT )
-> INDEX Scan USING i1 ON insegn i (cost =0.28..47.99 ROWS=2723 w=63)
   INDEX Cond : (ie. id_insegn > id)
```

- La condizione del JOIN è meno selettiva (le righe finali sono più di 14M).
- Il join viene fatto usando un loop (Nested Loop): ogni riga prodotta dal nodo Seq Scan si unisce con ogni riga prodotta dal nodo INDEX scan.
- L'unico Seq Scan che si potrebbe ottimizzare è quello evidenziato.

Explain - Struttura

Ottimizzazione query precedente

```
CREATE INDEX ie_aa ON Inserogato (annoaccademico varchar_pattern_ops);
ANALYZE Inserogato;
EXPLAIN SELECT *
FROM Insegn I JOIN Inserogato IE ON ie. id_insegn > i.id
WHERE ie. annoaccademico = '2013/2014';

QUERY PLAN
-------------------------------------------------
Nested Loop ( cost =82.65.. 391932.20 ROWS =14037065 width =641)
-> Bitmap Heap Scan ON inserogato ie (cost=82..4198 rows=5155 w=578)
Recheck Cond: ((annoaccademico)::TEXT = '2013/2014'::TEXT )
-> Bitmap INDEX Scan ON ie_aa (cost=0.00..81.08 ROWS=5155 w=0)
INDEX Cond: ((annoaccademico)::TEXT = '2013/2014':: TEXT )
-> INDEX Scan USING i1 ON insegn i (cost =0.28..48 ROWS =2723 w =63)
   INDEX Cond : (ie. id_insegn > id)
```

- Da 397.269 accessi a disco si è passati a 391.932!

Explain - Struttura

Altro tipo di JOIN: join mediante merge join

```
EXPLAIN SELECT * FROM t1, t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2 ;

QUERY PLAN
-------------------------------------------------------------
Merge JOIN (cost=198.11..268.19 ROWS=10 width=488)
Merge Cond: (t1.unique2 = t2.unique2 )
-> INDEX Scan USING t1_unique2 ON t1 (cost =0..656 ROWS =101..)
Filter : ( unique1 < 100)
-> Sort (cost =197.83..200.33 ROWS =1000..)
   Sort KEY: t2.unique2
-> Seq Scan ON t2 (cost =0.00..148.00 ROWS =1000..)
```

- Merge Join esegue il join ordinando prima le due tabelle rispetto gli attributi di join.
- t1 è già ordinata via indice.
- t2 non è ordinata, quindi c'è un nodo Sort.
- L'unico Seq Scan che si potrebbe ottimizzare è quello evidenziato: un indice eviterebbe il Sort.

Explain versione avanzata

- Il comando EXPLAIN ANALYZE mostra il piano di esecuzione, esegue la query senza registrare eventuali modifiche e mostra, infine, una stima verosimile dei tempi di esecuzione.

Explain versione avanzata

Explain con esecuzione

```
EXPLAIN ANALYZE SELECT * FROM Insegn WHERE id <1000 AND nomeins LIKE 'Algo %';

QUERY PLAN
-----------------------------------------------------------
Bitmap Heap Scan ON insegn (cost=18.40..134.62 ROWS=1..)
      (actual TIME=1.001..1.605 ROWS=1 loops=1)
  Recheck Cond: (id < 1000)
  Filter: (nomeins ~~ 'Algo %')
  ROWS Removed BY Filter: 814
  Heap Blocks: exact=90
  -> Bitmap INDEX Scan ON insegn_pkey (cost=0..18 ROWS=815)
       (actual TIME=0.928..0.928 ROWS=815 loops=1)
     INDEX Cond: (id < 1000)
Planning TIME: 1.243 ms
Execution TIME: 2.234 ms
```

- Per ogni nodo del piano: dettaglio righe rimosse, memoria usata e tempi di esecuzione in ms.
- Alla fine: stima tempo di pianificazione ed di esecuzione.

Explain versione avanzata

Explain con esecuzione di una query pesante

```
EXPLAIN ANALYZE SELECT * FROM insegn I, inserogato IE
WHERE IE.annoaccademico = '2013/2014' AND IE.id_insegn > I.id;

QUERY PLAN
-------------------------------------------------------------
Nested Loop (cost=0..397132 ROWS=14148708 width =638)
   (actual TIME=0.111..18713.811 ROWS=24457113 loops=1)
-> Seq Scan ON inserogato (cost=0..6059 ROWS =5196)
     (actual TIME=0.070..41.651 ROWS=5155 loops=1)
   Filter: (annoaccademico ='2013/2014')
     ROWS Removed BY Filter: 62862
-> INDEX Scan USING insegn_pkey ON insegn (cost =0..48.03 ROWS=2723)
   (actual TIME=0.010..1.790 ROWS=4744 loops =5155)
   INDEX Cond: (IE. id_insegn > id)
Planning TIME: 0.612 ms
Execution TIME: 21471.997 ms
```

- Il valore di loops indica per quante volte viene eseguito il nodo.
- Spesso il numero di righe dopo cost è diverso dal numero di righe dopo actual TIME. Quest'ultimo è più corretto.