

# PosgreSQL DBA TUNING



**Federico Campoli**

## **PosgreSQL DBA TUNING**

Federico Campoli

Prima Edizione

Pubblicato 2007

Copyright © 2007 PGHost di Federico Campoli

Opera rilasciata sotto licenza Creative Commons - **Attribuzione - Non commerciale - Non opere derivate 2.5**

### **Tu sei libero:**

- di riprodurre, distribuire, comunicare al pubblico, esporre in pubblico, rappresentare, eseguire e recitare quest'opera

### **Alle seguenti condizioni:**

- **Attribuzione.** Devi attribuire la paternità dell'opera nei modi indicati dall'autore o da chi ti ha dato l'opera in licenza.
- **Non commerciale.** Non puoi usare quest'opera per fini commerciali.
- **Non opere derivate.** Non puoi alterare o trasformare quest'opera, ne' usarla per crearne un'altra.

Ogni volta che usi o distribuisce quest'opera, devi farlo secondo i termini di questa licenza, che va comunicata con chiarezza. In ogni caso, puoi concordare col titolare dei diritti d'autore utilizzi di quest'opera non consentiti da questa licenza. Questa licenza non riduce o elimina in alcun modo i diritti morali dell'autore.

# Sommario

<b>1. Esecuzione di una query</b> .....	<b>1</b>
1.1. Il parser stage .....	1
1.2. Il query tree .....	1
1.3. L'executor.....	2
<b>2. Cenni di performance tuning</b> .....	<b>3</b>
2.1. Vacuum e analyze.....	3
2.1.1. XID Wraparound.....	3
2.2. Lettura di un piano di esecuzione .....	4
2.2.1. Gli operatori del piano d'esecuzione (cenni).....	4
2.3. tenere i dati vicino alla CPU .....	6
2.4. cluster e reindex .....	7
2.5. perche' non usa l'indice? .....	8

# Lista delle Figure

1-1. Query tree di una semplice select.....	1
---	---

# Capitolo 1. Esecuzione di una query

In questo primo capitolo analizzeremo il modo in cui viene eseguita una query, dal momento in cui raggiunge il server fino al momento in cui i dati sono effettivamente trasferiti all'applicazione client.

## 1.1. Il parser stage

Una qualsiasi query che viene mandata verso il server deve attraversare vari stadi prima di essere processata.

Che venga trasmessa attraverso odbc, libpq o librerie scritte ad hoc la query termina il suo viaggio sul backend del processo postmaster, il processo postgres che si occupa della gestione della query.

La prima operazione che viene eseguita è l'invio della query al parser che ne verifica la sintassi e la riscrive in quello che viene chiamato "parse tree".

Il parse tree è un insieme di elementi che descrive in maniera "cruda" il significato della query tenendo conto esclusivamente della corretta sintassi.

## 1.2. Il query tree

Se il parse tree non contiene errori, questi viene rimandato al parser che ne esegue l'interpretazione e la trasformazione semantica per determinare in maniera univoca quali relazioni e operatori sono referenziati nella query.

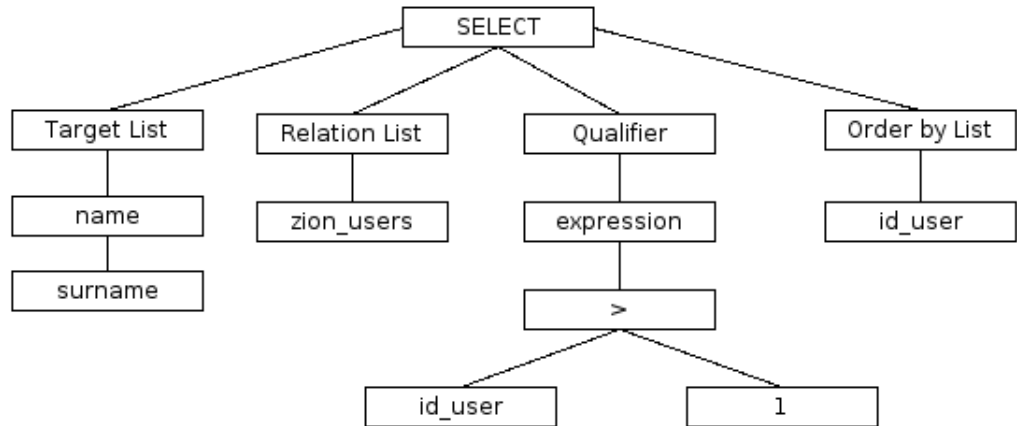
Al termine di questo processo viene prodotto il "query tree", ovvero un insieme di elementi che in maniera univoca identifica il significato della query e gli oggetti toccati da questa.

La separazione netta tra la fase di verifica sintattica e di analisi semantica è voluta.

Infatti durante l'analisi semantica è necessario eseguire delle ricerche sul catalogo di sistema che richiedono l'apertura di una transazione.

Figura 1-1. Query tree di una semplice select

**SELECT name, surname FROM zion\_users WHERE id\_user>1 ORDER BY id\_user;**



### 1.3. L'executor

Una volta definito il query tree il query planner esegue una scansione del query tree per trovare tutti i possibili piani d'esecuzione per risolvere la query.

Il planner assegna alle varie operazioni un costo che viene misurato in unita' I/O piu' una percentuale maggiorativa di questo costo legata al tempo di processore richiesto.

Il piano la cui stima di costo e' minore di tutti viene scelto per la risoluzione della query.

Non sempre il piano scelto corrisponde al piano migliore.

# Capitolo 2. Cenni di performance tuning

## 2.1. Vacuum e analyze

Vacuum vuol dire letteralmente passare l'aspirapolvere. Il comando **VACUUM** compatta le pagine dati eliminando le dead tuples e, se lanciato con la clausola **analyze** aggiorna le statistiche delle relazioni necessarie all'ottimizzatore per produrre piani d'esecuzione soddisfacenti.

Il **VACUUM** puo essere lanciato con la clausola **full** e **analyze**. Il **VACUUM** lanciato con la clausola **analyze** opera prima un vacuum e successivamente esegue un **analyze** della tabella per recuperare informazioni quali numero di righe, istogrammi di occorrenze valori in un indice etc.

Il **VACUUM full** esegue una compattazione molto piu' radicale aprendo una transazione per operare in maniera consistente e spostando le righe da una pagina all'altra per ottenere il massimo risultato di compattazione possibile. Il **VACUUM FULL** inoltre previene il rischio di **XID wraparound** come vedremo in seguito.

### 2.1.1. XID Wraparound

Operativamente, ogni operazione di tipo DML effettua una transazione autocommittante. Ogni transazione riceve un identificativo numerico intero a 32bit chiamato **XID**. Ogni database del cluster ha un contatore separato dei propri **XID**.

La semantica transazionale di PostgreSQL segue questa logica: *Tutti gli XID che sono maggiori di quello corrente sono considerati nel futuro e quindi invisibili alla transazione corrente.*

Un contatore numerico unsigned a 32 bit esegue un wraparound dopo 4.294.967.296 scatti. Poiche' gli **XID** maggiori dello **XID** corrente sono considerati "nel futuro" al wraparound tutte le transazioni committate nel passato si ritroverebbero improvvisamente nel futuro e quindi invisibili alla transazione corrente. Al wraparound i dati pur essendo ancora presenti nel database sono inaccessibili!

Fino alla versione 7.2 l'unica soluzione al problema dello **XID wraparound** era quello di fare un **initdb** della directory data ogni 4 miliardi di transazioni. Dalla 7.3 in poi e' sufficiente un **VACUUM FULL** ogni 2 miliardi di transazioni per impedire l'**XID wraparound** Il comando **VACUUM** non full, all'avvicinarsi del limite, scrive dei warning nel log di esercizio quando ci sono database del cluster che si trovano a meno di 500 milioni di transazioni dal wraparound failure. A meno di 10 milioni di transazioni dal wraparound failures questo warning viene inviato per ogni transazione. A meno di 1 milione il postmaster termina spontaneamente ad ogni transazione

Il vacuum periodico risolve il problema poiche' PostgreSQL distingue un tipo speciale di XID chiamato FrozenXID. Questo XID e' sempre considerato piu' vecchio di qualsiasi XID normale. Gli XID normali esistono in uno spazio circolare senza fine grazie al metodo di confronto (modulo-2^31). Una volta che una riga e' stata creata questa riga appare nel passato per i 2 miliardi successivi di transazioni indipendentemente dal suo valore. All'avvicinarsi del limite di 2 miliardi di transazioni, che porterebbe quindi la riga nel "futuro" facendola svanire nel nulla digitale, il VACUUM riassegna a questa riga un FrozenXID che resta assegnato alla riga cosi' preservata dal wraparound finche' questa riga non viene cancellata.

## 2.2. Lettura di un piano di esecuzione

La clausola explain anteposta ad una query determina la visualizzazione del piano d'esecuzione stimato da PostgreSQL per la query indicata. Dalla lettura del piano d'esecuzione e' un'operazione fondamentale per eseguire un tuning ottimale.

Un piano d'esecuzione si legge a partire dalla riga piu' indentata a destra e piu' in basso. La lettura prosegue quindi da destra verso sinistra e dal basso verso l'alto.

```
postgres=# explain select * from test where objid=10749 limit 10;
QUERY PLAN
-----
Limit  (cost=11.93..40.53 rows=10 width=25)
-> Bitmap Heap Scan on test  (cost=11.93..1353.04 rows=469 width=25)
    Recheck Cond: (objid = 10749::oid)
-> Bitmap Index Scan on idx_test  (cost=0.00..11.82 rows=469 width=0)
    Index Cond: (objid = 10749::oid) (5 rows)
```

Questo piano d'esecuzione ad esempio indica che abbiamo imposto una condizione di where che verifica l'utilizzo di un indice *Index Cond*. La risoluzione della condition avviene attraverso un *Bitmap Index Scan* che viene poi completato con un *Bitmap Heap Scan* avente come *Recheck Cond* la stessa *Index Cond* della fase precedente.

Le singole voci di un piano di esecuzione hanno la seguente struttura:

Operatore (cost=COST\_MIN..COST\_MAX rows=NUM\_ROWS width=ROW\_WIDT)

I numeri corrispondenti a cost sono il costo minimo e massimo stimato. Il valore corrispondente a rows indica il numero di righe stimate dall'ottimizzatore determinato dalle statistiche. L'ultimo numero indica quanto ampia, in byte, e' una singola riga recuperata.

## 2.2.1. Gli operatori del piano d'esecuzione (cenni)

Un operatore e' una trasformazione che l'executor applica ad un insieme di righe.

Questo insieme, durante il suo movimento lungo il piano di esecuzione viene trasformato ed adattato a seconda di quanto scritto nella query.

Il costo minimo e massimo stimati indicano quanto l'ottimizzatore stimi il costo dell'operatore.

Il valore di rows e width indicano rispettivamente il numero di righe e l'ampiezza in byte delle righe che vengono recuperate dall'operatore.

Come si puo' notare tutte queste informazioni sono delle stime ricavate dalle statistiche interne del database che vengono aggiornate con il comando vacuum o con analyze.

Pertanto, per ottenere piani d'esecuzione soddisfacenti e' fondamentale prevedere la manutenzione periodica del cluster database.

Inserendo la clausola ANALYZE dopo EXPLAIN il piano di esecuzione non viene stimato. La query viene realmente eseguita e il suo piano d'esecuzione, con i dati non piu' stimati ma reali, viene visualizzato a video.

Di seguito sono elencati e descritti alcuni dei piu' comuni operatori dell'executor.

### 2.2.1.1. sequential\_scan

Accetta come parametro di input una tabella e restituisce le sole righe specificate dalla clausola di where. Ha il vantaggio di lavorare in stream, ovvero durante l'esecuzione del sequential scan le righe ritrovate sono gia' disponibili per il successivo operatore oppure per la restituzione al backend. Questo operatore lavora per sottrazione. Legge ogni riga della tabella ed eventualmente scarta quelle che non corrispondono alla clausola di where.

### 2.2.1.2. index\_scan

Accetta come parametro di input un indice definito su di una o piu' colonne di tabella e restituisce le sole voci di indice specificate dalla clausola di where. Ha il vantaggio di lavorare per range, ovvero definito un valore massimo e un minimo nella where condition l'indice viene trasversato solo per quei valori. A differenza del seq\_scan l'index\_scan restituisce i dati gia' ordinati.

### **2.2.1.3. bitmap\_scan**

Esegue una lettura sequenziale delle pagine index ma senza seguire l'indice. La lettura avviene in maniera simile ad un sequential scan. Una volta in memoria gli indici vengono trasformati in bitmap e sommati per trovare le occorrenze che soddisfano il filtro.

### **2.2.1.4. merge join**

Accetta due set di righe in ingresso e restituisce un'unica riga prodotta dall'affiancamento delle righe in ingresso. Necessita che i dati siano ordinati.

### **2.2.1.5. sort**

Accetta come input un set di dati e li restituisce ordinati. Non lavora in stream, ovvero richiede l'intero set per poter ordinare.

### **2.2.1.6. limit**

Accetta come input un set di dati e restituisce solo i primi x specificati nella clausola di avvio.

### **2.2.1.7. unique**

Accetta come input un set di dati e scarta i valori duplicati.

## **2.3. tenere i dati vicino alla CPU**

Per eseguire un tuning ottimale bisogna tener conto di due aspetti fondamentali.

Il primo aspetto è da tenere in considerazione e' l'uso ottimale dell'intero sistema, cpu, memoria e dischi.

Il secondo aspetto di cui tener conto e' l'ottimizzazione delle query impiegate nell'applicazione.

Il problema che viene costantemente sottovalutato quando si crea un'applicazione e' la struttura a cipolla che compone un moderno computer.

Si parte dallo strato piu' interno, i registri di CPU, si passa alla cache della CPU si arriva alla cache del kernel e infine al sottosistema disco.

I dati, per essere processati devono attraversare, partendo dallo storage su disco, quattro strati prima di arrivare al processore che li dovra' elaborare.

E' quindi pacifico che piu' i dati sono vicini al processore tanto piu' performanti saranno le operazioni di lettura.

Il tuning sui registri e sulla cache della CPU e' fuori dalla portata di un DBA poiche' gli eseguibili sono gia' prodotti da compilatori che producono codice ottimizzato per sfruttare l'architettura su cui gira il database.

Per ottenere quindi un tuning efficace bisogna tener conto di quanta ram e' installata sul sistema e come la distribuzione dei programmi e del sistema operativo impegnano la sua memoria.

La ram di un sistema in esercizio contiene le seguenti cose

- Programmi in esecuzione
- Dati dei programmi in esecuzione
- Lo shared buffer di PostgreSQL
- Il buffer del kernel
- Il kernel

Un tuning ottimale della ram deve ottenere come risultato di avere la maggior parte delle informazioni del database in memoria senza toccare in senso negativo le altre aree impegnate della RAM.

## **2.4. cluster e reindex**

I comandi cluster e reindex possono avere un impatto non indifferente sulle performance di una tabella.

PostgreSQL allo stato attuale non e' in grado di recuperare dati di heap da un indice, quindi all'accesso di un heapindex corrisponde un ulteriore seek su disco per il recupero dei dati relativi alla riga indicizzata.

PostgreSQL inserisce i dati nelle tabelle in maniera sequenziale. Gli stessi update non fanno altro che inserire una nuova riga in fondo alla tabella. Di contro invece l'indice viene costruito in maniera ordinata.

Pertanto esiste, normalmente, una mancanza di ordinamento tra le pagine di indice e le pagine dati.

Quando si va a leggere una tabella attraverso un `index_scan` i blocchi di indice vengono letti in maniera ordinata.

Quando viene trovata una corrispondenza PostgreSQL carica nello `shared_buffer` la pagina corrispondente alla corrispondenza ed estrae la riga trovata.

Andando avanti nella lettura dell'indice magari la successiva corrispondenza si trova su di un'altra pagina che deve essere caricata anch'essa in memoria per la lettura della riga.

In questo modo ogni volta che viene trovata una occorrenza nell'indice si rischia di dover caricare una pagina nello `shared_buffer` producendo un elevato flusso I/O a causa del buffer manager.

Il comando `cluster` semplicemente riordina le righe di tabella secondo l'ordinamento dell'indice in modo che esista una corrispondenza uno a uno tra una pagina indice e una pagina dati in modo da avere nel buffer tutti i dati di riga corrispondenti ai dati di indice.

`REINDEX` ricostruisce un indice usando i dati della tabella. Reindex puo' tornare utile in varie situazioni.

- Riparazione di un Indice corrotto con dati non piu' consistenti con quelli di tabella.
- Ricostruzione di indice "bloated". Un indice bloated contiene pagine index vuote o quasi vuote. Il bloat di indice era un problema piuttosto comune con la versione 7.4. Dalla 8.0 in poi e' stato reso piu' funzionale il riciclo delle pagine index da parte di `VACUUM` ma il problema in maniera minore sussiste ancora.
- Un parametro di storage e' stato modificato ed e' necessario ricostruire l'indice affinche' questo venga sfruttato a pieno.
- Una creazione di indice avviata con la clausola `CONCURRENTLY` ha fallito lasciando l'indice invalidato. Reindex ricostruisce l'indice da zero.

**Importante:** `REINDEX` non lavora in maniera concorrente. Pertanto se si vuole costruire un indice senza interferire con l'attivit  del database e' necessario dropare l'indice invalido e ricrearlo con la clausola `CONCURRENTLY`.

## 2.5. perche' non usa l'indice?

La domanda piu' classica che puo' capitare analizzando i piani di esecuzione di un database e' "perche' non usa l'indice?".

I fattori che spingono l'ottimizzatore a usare un index scan piuttosto che un sequential\_scan sono molteplici.

A titolo d'esempio, un full index scan, e' sempre piu' costoso di un sequential scan.

L'indice viene quindi usato in determinate condizioni sempre dipendenti dal costo stimato dall'ottimizzatore.

A titolo d'esempio creiamo una tabella di test contenente i dati di pg\_class.

```
CREATE TABLE test
(
  id serial NOT NULL,
  testo character varying(100),
  CONSTRAINT pk_test PRIMARY KEY (id)
)
WITHOUT OIDS;

INSERT INTO test (testo) SELECT relname FROM pg_class; -- insert di 191 righe
VACUUM FULL test;
```

L'explain di una query di ricerca con filtro id pari a 3 rivela che viene adoperato un sequential scan invece di un index scan.

```
postgres=# explain select testo from test where id=3;
QUERY PLAN
-----
Seq Scan on test (cost=0.00..4.39 rows=1 width=218)
  Filter: (id = 3) (2 rows)
```

Proviamo a disabilitare i sequential scan con il comando set

```
postgres=# SET enable_seqscan=off;
SET
postgres=# explain SELECT TESTO FROM test where id=3;
QUERY PLAN
-----
Index Scan using pk_test on test (cost=0.00..5.21 rows=1 width=218)
  Index Cond: (id = 3) (2 rows)
```

Il costo del sequential scan e' di 4.39 mentre quello dell'index scan e' di 5.21.

A questo punto aumentiamo il numero di righe nella tabella. e rifacciamo vacuum

```
postgres=# EXPLAIN SELECT TESTO FROM test WHERE id=3;
QUERY PLAN
-----
Index Scan using pk_test on test (cost=0.00..6.01 rows=1 width=218)
  Index Cond: (id = 3) (2 rows)
```

In questo caso il costo dell'index\_scan e' simile a quello precedente.

Disabilitiamo l'index scan e riproviamo l'explain.

```
postgres=# explain select testo from test where id=3;
QUERY PLAN
-----
Bitmap Heap Scan on test (cost=2.00..6.02 rows=1 width=218)
  Recheck Cond: (id = 3) > Bitmap Index Scan on pk_test
    (cost=0.00..2.00 rows=1 width=0)
    Index Cond: (id = 3) (4 rows)
```

In maniera molto furba l'ottimizzatore esegue un bitmap index scan dal costo di poco superiore all'index scan.

Disabilitiamo anche il bitmap index scan:

```
postgres=# explain select testo from test where id=3;
QUERY PLAN
-----
Seq Scan on test (cost=0.00..7723.60 rows=1 width=218)
  Filter: (id = 3) (2 rows)
```

A questo punto l'ottimizzatore deve scegliere per forza il sequential scan che ha un costo 3 ordini di grandezza piu' elevato rispetto all'index scan.

Da questo semplice esempio si evince che, per tabelle piccole il sequential scan e' sempre piu' conveniente poiche', in genere, con un singolo accesso al disco permette di portare l'intera tabella nello shared buffer.

Per tabelle piu' grandi il costo del seq\_scan cresce mentre quello dell'index\_scan, se limitato a valori singoli resta piu' o meno costante.