Basi di Dati

Uso di informazioni statistiche e partizionamento delle tabelle per incrementare le performances in PostgreSQL

di Matteo Bertini

Email: matteo@naufraghi.net
Web: http://www.slug.it/naufraghi/

25 Agosto 2006

Indice

1 Analisi dell'archivio

L'archivio è composto dai post di circa duemila utenti prelevati da del.icio.us, ogni post è costituito da:

user. utente che ha creato il post,

href. indirizzo internet inserito in archivio dall'utente,

tags. eventuale list di "tag", etichette che, secondo l'utente, caratterizzano l'indirizzo internet.

1.1 Distribuzioni dei dati

Per avere un'idea della struttura dell'archivio è possibile rappresentare alcune distribuzioni.

Ad esempio, un primo interesse è quello di capire quanti indirizzi sono presenti in archivio per ogni utente.

Per far questo ho proceduto prima contando gli indirizzi per utente e poi raggruppando gli utenti per numero di indirizzi. L'immagine seguente rappresenta il grafico di tale conteggio.

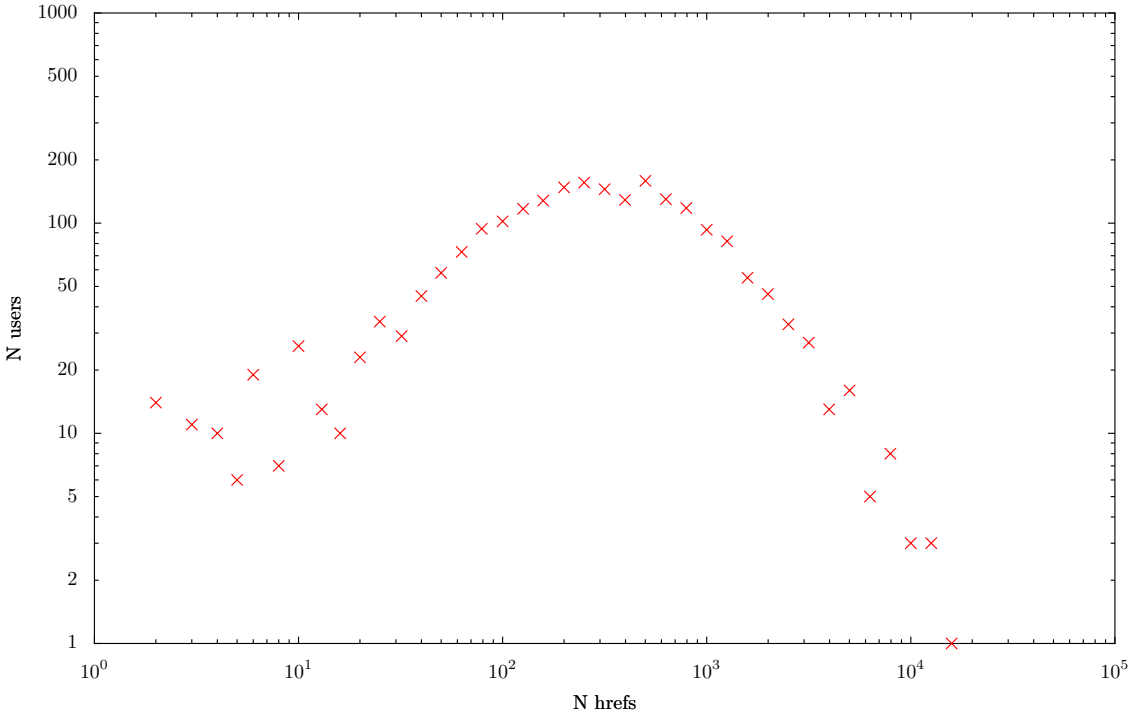


Figura 1. Numero di utenti raggruppati per numero di indirizzi.

Da osservare come entrambi gli assi siano in scala logaritmica.

Possiamo osservare come la maggior parte degli utenti abbia in archivio un numero di indirizzi nell'intervallo $10^2 - 10^3$ e che relativamente pochi utenti abbiano un numero sensibilmente superiore o inferiore ai limiti di tale intervallo.

Analogamente ho proceduto raggruppando gli indirizzi per numero di utenti, e rappresentando il grafico di tale conteggio.

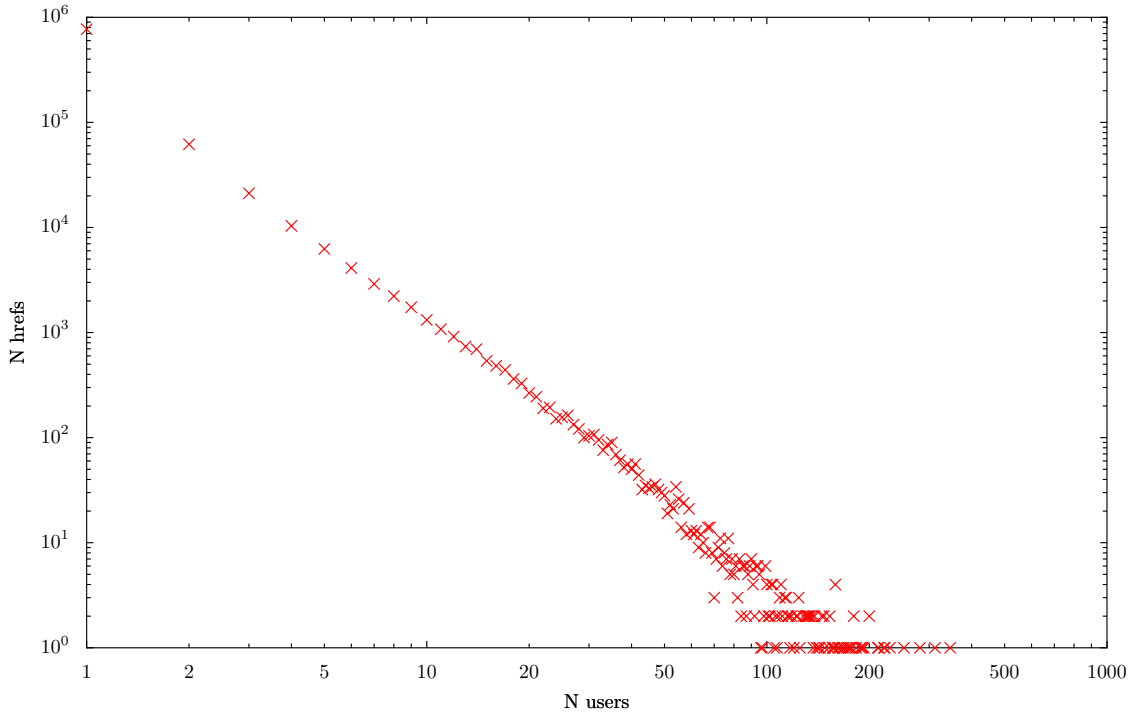


Figura 2. Numero di indirizzi raggruppati per numero di utenti.

Possiamo osservare come tale configurazione ricordi la distribuzione di Zipf, o in generale un comportamento che si adatta al principio di Pareto, in cui la maggioranza degli indirizzi internet è "supportata" da un numero esiguo di utenti, mentre un numero esiguo di indirizzi catalizza l'attenzione della maggioranza degli utenti.

2 Introduzione al partizionamento

Il partizionamento è una tecnica presente nei database più evoluti che permette al progettista di definire a priori una suddivisione dei dati di una tabella in più sotto-tabelle.

Esempi tipici di questa tecnica sono le suddivisioni in blocchi temporali. Ad esempio un sistema che conserva dei log potrebbe partizionare periodicamente l'archivio in blocchi temporali, in modo che le prestazioni dell'inserimento non degradino nel tempo (al crescere indefinito della tabella), ma che invece rimangano più o meno costanti mediante la creazione di una nuova sotto-tabella per i nuovi log ogni periodo prestabilito di tempo[1].

2.1 Partizionamento in PostgreSQL

Il partizionamento in PostgreSQL è basato sull'ereditarietà[2]. In PostgreSQL è possibile definire una tabella come "erede" di una tabella preesistente. La tabella "figlia" eredita tutti i campi della tabella "madre" e può avere campi aggiuntivi.

L'idea generale per implementare il partizionamento in PostgreSQL è di creare una tabella "madre" con tutti i campi necessari (ma senza indici, perché destinata a rimanere vuota), ed ereditare da questa le sotto-tabelle che effettivamente andranno a contenere i dati.

Inserendo opportuni `RULES` sulla tabella "madre" è possibile filtrare in modo trasparente tutte le operazioni di `INSERT`, `SELECT`, `UPDATE` e `DELETE` in modo da essere propagate alla sotto-tabella opportuna.

Una volta creata una struttura che si adatti all'uso dei dati, attivando l'opzione `constraint_exclusion` l'ottimizzatore di query è in grado di limitare la query alle sole tabelle con i `CONTRAINT` compatibili con la clausula `WHERE` fornita.

2.1.1 Struttura delle tabelle

Per implementare il partizionamento è necessaria una certa confidenza con gli automatismi offerti da PostgreSQL, per questo è disponibile una completa guida online[3].

Il nostro test si concentra sulla creazione di un partizionamento basato su un campo che conteggia il numero di occorrenze di un dato elemento.

Riducendo al minimo la complessità della tabella, i campi scelti sono:

id1. chiave primaria della tabella tag,

id2. chiave primaria della tabella href,

nusers. numero di utenti che usano l'associazione tag-href.

Definiamo per prima una tabella `prova1`:

```
CREATE TABLE prova1
(
    id1 INT8,
    id2 INT8,
    nusers INT8,
    CONSTRAINT prova1_pkey PRIMARY KEY (id1, id2)
```

1. http://www.postgresql.org/docs/8.1/interactive/ddl-partitioning.html
2. http://www.postgresql.org/docs/8.1/interactive/ddl-inherit.html
3. http://www.postgresql.org/docs/8.1/interactive/index.html

```
);
```

definiamo quindi una tabella "madre", `prova2`, simile a `prova1` ma senza `CONSTRAINT`:

```
CREATE TABLE prova2
(
    id1 INT8,
    id2 INT8,
    nusers INT8
);
```

seguita dalle due tabelle partizionate `prova2_head` e `prova2_tail`:

```
CREATE TABLE prova2_head
(
    CONSTRAINT prova2_head_pkey PRIMARY KEY (id1, id2),
    CONSTRAINT head CHECK (nusers > 1)
) INHERITS prova2;
```

La prima ha un controllo ed ammette i soli elementi con `nusers > 1`, la seconda ammette solo elementi con `nusers <= 1` (in effetti solo 1, ma usando l'operatore "<=" si evitano i problemi del confronto in uguaglianza tra tipi numerici diversi).

```
CREATE TABLE prova2_tail
(
    CONSTRAINT prova2_tail_pkey PRIMARY KEY (id1, id2),
    CONSTRAINT tail CHECK (nusers <= 1)
) INHERITS prova2;
```

Da osservare, per il partizionamento è consigliabile che i `CHECK` suddividano il dominio in insiemi contigui ma disgiunti.

2.1.2 RULES per INSERT e UPDATE

Una volta definite le tabelle, nel caso della tabella partizionata dobbiamo istruire PostgreSQL in modo da inserire i dati nella sotto-tabella corretta.

In generale la tabella `prova2` è destinata a rimanere vuota. I dati devono essere rediretti, coerentemente con i `CHECK`, alle due sotto-tabelle.

Gestiamo inizialmente l'evento `ON INSERT`:

```
CREATE OR REPLACE RULE prova2_insert_head AS
    ON INSERT TO prova2 WHERE new.nusers > 1 DO INSTEAD
        INSERT INTO prova2_head (id1, id2, nusers)
            VALUES (new.id1, new.id2, new.nusers);
```

Questo `RULE` redirige gli inserimenti con `nusers > 1` nella tabella `head`, analogamente per la tabella `tail` abbiamo:

```
CREATE OR REPLACE RULE prova2_insert_tail AS
    ON INSERT TO prova2 WHERE new.nusers <= 1 DO INSTEAD
        INSERT INTO prova2_tail (id1, id2, nusers)
            VALUES (new.id1, new.id2, new.nusers);
```

Minimamente più complessa la gestione degli update, nel caso della tabella partizionata tutte le volte che un elemento passa da `nusers <= 1` a `nusers > 1`, o viceversa, l'`UPDATE` comporta l'aggiornamento di due tabelle. Infatti è necessario cancellare l'elemento da `tail` ed inserirlo in `head` al crescere di `count` e cancellare in `head` per inserire in `tail` al decrescere.

```
CREATE OR REPLACE RULE prova2_update_to_head AS
    ON UPDATE TO prova2 WHERE old.nusers <= 1 AND new.nusers > 1 DO INSTEAD (
        INSERT INTO prova2_head (id1, id2, nusers)
            VALUES (old.id1, old.id2, new.nusers);
        DELETE FROM prova2_tail
            WHERE prova2_tail.id1 = old.id1 AND prova2_tail.id2 = old.id2;
    );
```

Analoga forma per l'`UPDATE` nel senso opposto.

```
CREATE OR REPLACE RULE prova2_update_to_tail AS
    ON UPDATE TO prova2 WHERE old.nusers > 1 AND new.nusers <= 1 DO INSTEAD (
        INSERT INTO prova2_tail (id1, id2, nusers)
            VALUES (old.id1, old.id2, new.nusers);
        DELETE FROM prova2_head
            WHERE prova2_tail.id1 = old.id1 AND prova2_tail.id2 = old.id2;
    );
```

2.1.3 Stored procedures

In un progetto complesso può dimostrarsi saggio astrarre dalla struttura dell'archivio fin dalle prime fasi dello sviluppo, per questo è possibile usare le Stored Procedures. In PostgreSQL è possibile scrivere procedure in vari linguaggi, dal semplice SQL, al linguaggio procedurale integrato PL/pgSQL, fino a numerosi linguaggi di scripting (Perl, Tcl e Python), nonché procedure compilate, scritte in C o in Java.

Data la mia esperienza con il linguaggio Python, ho scelto di scrivere le procedure in PL/Python. La versione del linguaggio procedurale fornita con PostgreSQL 8.1 non permette però la restituzione di tipi complessi (non è attivo il `RETURN SETOF TYPE` per tipi complessi come `ROW`), pertanto nei miei test ho usato una versione di PL/Python compilata dal CVS del progetto.

Purtroppo l'implementazione attuale del partizionamento in PostgreSQL non permette di ereditare gli indici, pertanto anche definendo un indice univoco sul campo `(id1, id2)` della tabella `prova2`, al momento dell'esecuzione di una query secca su chiave univoca verrebbero sempre analizzate entrambe le tabelle `prova2_head` e `prova2_tail`, anche se la query potrebbe correttamente fermarsi al primo elemento trovato. Per fortuna, nota l'attuale situazione, un semplice `LIMIT 1` risolve comunque il problema, infatti se il profiler "preferisce" la tabella di testa a quella di coda (ipotesi spesso corretta date diverse dimensioni), la tabella di testa verrà analizzata per prima e la tabella di coda verrà considerata solo nel caso la prima query fallisca[4].

Il caso delle query secca non è però molto frequente nell'applicazione che sto sviluppando, in particolare le query più frequenti sono quelle che richiedono la risoluzione di una sola delle due chiavi che compongono l'indice (`id1` ad esempio), e la restituzione del "vettore" avente come componenti la seconda chiave `id2` e come modulo `nusers`.

Ancora più frequenti sono le query che chiedono questo stesso vettore, ma solo se il modulo è strettamente maggiore di 1.

Le procedure necessarie per la gestione della tabella sono due:

`insert_or_update(tbl TEXT, id1 INT4, id2 INT4, nusers INT8, uop TEXT)`. La funzione di inserimento grazie all'uso trasparente dei `RULES` è identica per le due tabelle, e agisce su una o l'altra tabella in funzione del parametro `tbl`. Il parametro `uop` indica il tipo di aggiornamento, `'+'` per incrementare, `'-'` per decrementare e `'='` per impostare al valore `nusers` fornito.

`get_vec(id1 INT4, common TEXT)`. La funzione restituisce gli elementi della tabella che hanno `id1` uguale a quello fornito e `nusers > 1` quando `common != 0`. Anche in questo caso, grazie all'uso trasparente dell'ereditarietà la funzione è identica per le due tabelle.

4. Grazie alla collaborazione della mailing-list della Comunità Italiana PostgreSQL: `http://www.psql.it`.

In allegato il dump dello schema del database con i sorgenti delle procedure.

3 Modello del test

Il test si basa sull'inserimento di dati generati in modo automatico secondo una distribuzione esponenziale.

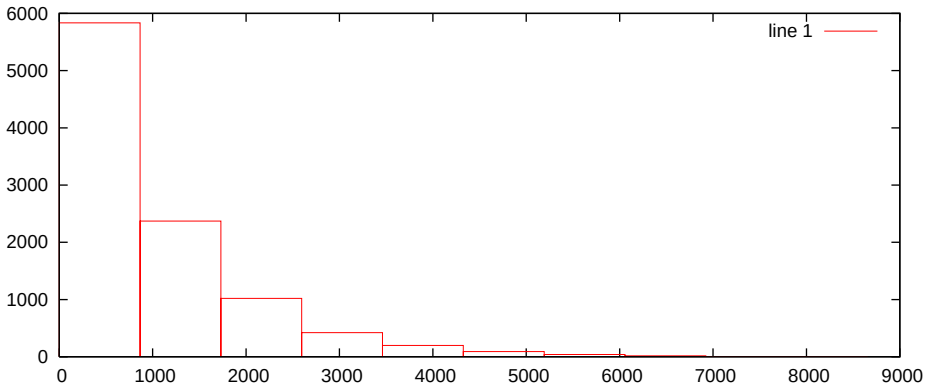
```
octave> exponential_rnd(1, 1, 10)
```

(0.7521 0.3472 0.5635 0.2971 0.09394 3.205 1.242 1.433 1.51 1.976)

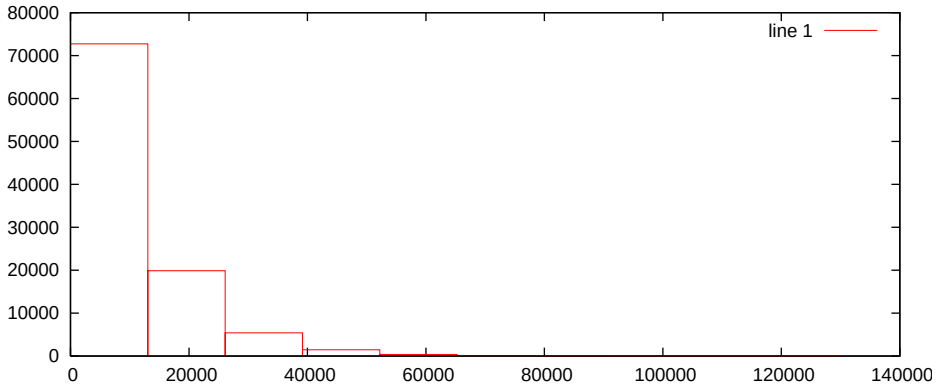
Gli indici corrispondenti ai tag (`id1`) vengono generati prendendo il valore intero del numero random generato dalla funzione `exponential_rnd(1)` moltiplicato per 1000. Gli indici associati agli href (`id2`) hanno invece una base di 10.000, un ordine di grandezza superiore.

Possiamo avere un'idea della distribuzione degli indici rappresentando un istogramma:

```
octave> __gnuplot_set__("size 1.0, 0.6");
        hist(1000 * exponential_rnd(1, 1, 10000)); tmreplot(); #tags
```




```
octave> hist(10000 * exponential_rnd(1, 1, 100000)); tmreplot(); #hrefs
```




```
octave>
```

Le collisioni in archivio saranno un numero ancora più limitato, perché legato al presentarsi più volte della stessa coppia `(id1, id2)`. Ipotesi sperimentalmente validata dai grafici presentati nella prima sezione.

Il test procede eseguendo serie successive di 10.000 inserimenti e 1000 query, di cui 1/6 = 17% a base allargata (log del numero di inserimenti). Le query hanno la stessa distribuzione degli inserimenti, ma sono più complesse e restituiscono tutte le righe associate ad un dato `id1`.

Riassumendo, in inserimento abbiamo:

- `id1 = 1000 * exponential_rnd(1)`
- `id2 = 10 * 1000 * exponential_rnd(1)`

In ricerca abbiamo:

- `id1 = 1000 * exponential_rnd(1)`
- il 17% delle query avviene sulla totalità della tabella (anche `nusers = 1`)

Dai dati sperimentali si cerca la dimensione limite dell'archivio per cui il partizionamento porti vantaggi in prestazioni rispetto alla struttura a tabella singola.

4 Risultati

I risultati del benchmark dimostrano che sfruttare informazioni statistiche note a priori è una importante risorsa nella progettazione di un database, in particolare se il database dovrà essere usato in una applicazione web, dove il numero di utenti può crescere rapidamente, e dove la bassa latenza della risposta è un parametro essenziale per il successo.

Rappresentiamo il grafico delle query, abbiamo rappresentato il tempo necessario al completamento di 10.000 operazioni in funzione del numero di inserimenti compiuti in archivio. Nell'ultima parte del grafico la tabella di testa contiene circa il 10% dei dati presenti.

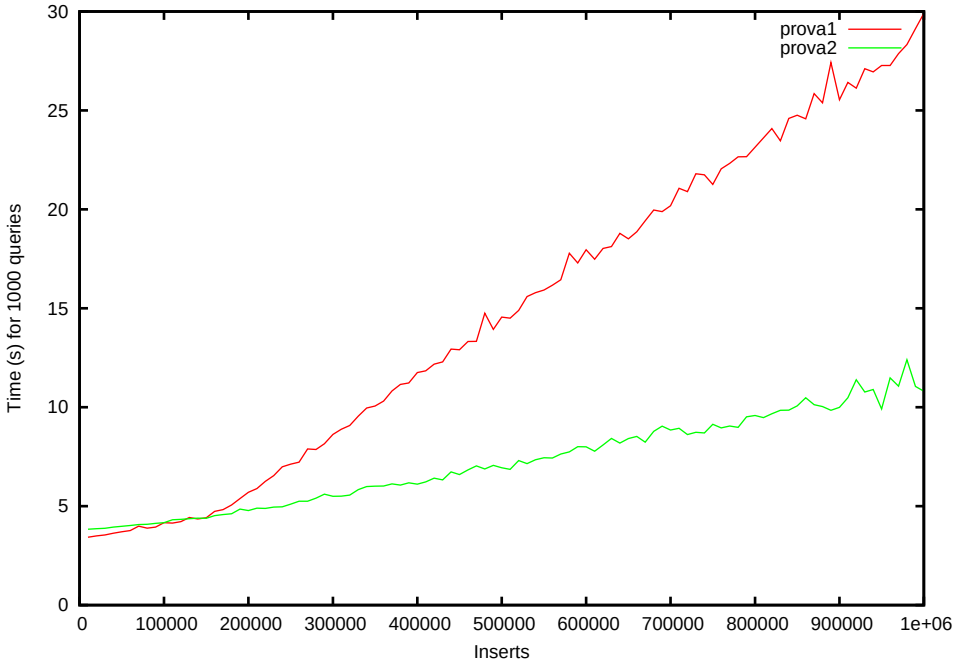


Figura 3. Tempo per 1000 query al crescere dell'archivio

Il comportamento delle query in inserimento è molto diverso. Nelle fasi iniziali l'inserimento dei `RULES` ha degli effetti negativi sulle prestazioni, però possiamo osservare che al crescere delle dimensioni dell'archivio, il prezzo costante della gestione dei `RULES` diventa via via meno preponderante sul prezzo crescente della gestione degli indici.

Infatti, rappresentando il grafico del tempo di inserimento (10.000 inserimenti) in funzione della dimensione dell'archivio, possiamo osservare che attorno al milione di righe le prestazioni della tabella partizionata superano quelle dalla tabella unica.

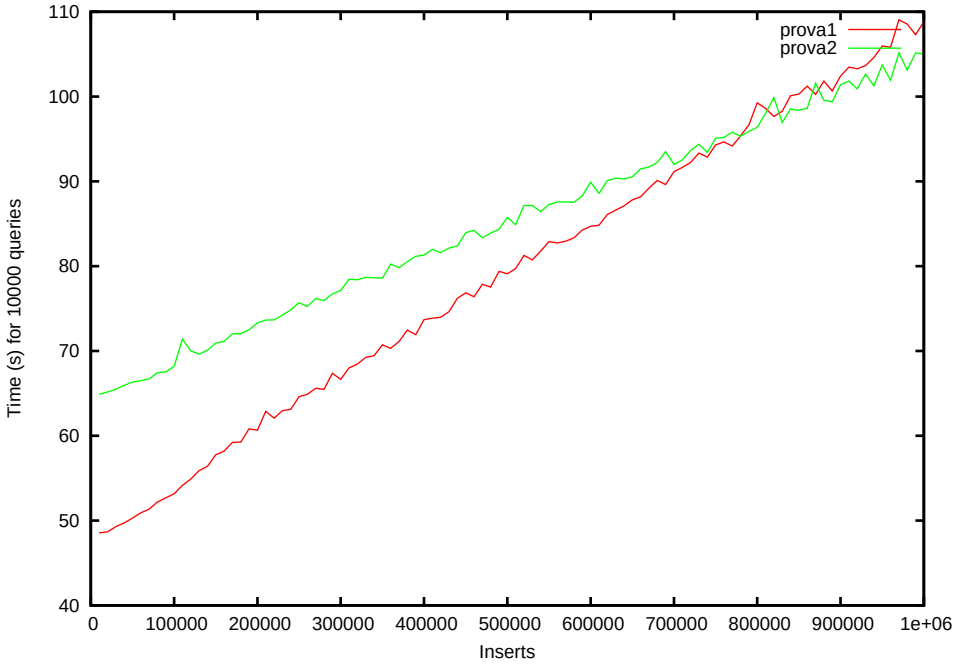


Figura 4. Tempo di inserimento in funzione del numero totale di inserimenti.

4.1 Analisi dei risultati

Abbiamo visto che, almeno con le ipotesi statistiche finora presentate, è possibile partizionare una tabella in modo da migliorare le prestazioni sia in inserimento che in lettura.

Ovviamente benchmark fatti con dati generati in modo automatico possono creare fragili illusioni, ma comunque danno spunto a nuovi approcci per tutti i casi in cui la caratterizzazione statistica dell'uso del database è nota e piuttosto stabile.

Alternative da inserire nel benchmark potrebbero essere l'uso di un unica tabella ma con due indici organizzati secondo lo stesso principio delle partizioni, o anche l'incremento del numero delle partizioni fino ad identificare il partizionamento ottimale per la dimensione dell'archivio.

Il vantaggio del partizionamento è che tabelle fisicamente diverse possono più facilmente essere distribuite su macchine diverse, questo nell'ottica odierna di progettare sistemi facilmente parallelizzabili.

5 Conclusioni

Abbiamo visto che usando il partizionamento delle tabelle è possibile ottimizzare la struttura di un archivio PostgreSQL in modo trasparente all'implementazione, e ridurre anche del 60% il tempo delle query più comuni. Questo semplicemente grazie ad informazioni statistiche note a priori.

Riassumendo, quando i dati hanno una distribuzione tipo Zipf, partizionando la tabella in due parti, abbiamo:

in lettura. le prestazioni della tabella partizionata superano quelle della tabella unica attorno alle 150.000 righe,

in scrittura. le prestazioni delle tabella partizionata superano quelle della tabella unica attorno alle 800.000 righe.

La gestione del partizionamento delle tabelle è una tecnologia relativamente nuova e non molto ottimizzata in PostgreSQL, questi sono indizi che lasciano sperare in un ulteriore incremento delle prestazioni, anche solo attendendo le future versioni del database.