Hash Index vs B-tree Index

Table of Contents:

What is Indexing?

To understand what indexing is, first let us consider a sample Database having a table called Employee which has Name, Age and Sex as the columns. Assume that this table has thousands of rows. Suppose we want to find all employees who have name as “Ram”.

Without an index, the Databse software would have to sequentially scan every single row in the table to see if the employee name matches 'Ram'. This is called a full table scan.

The whole point of indexing is to speed up search queries by cutting down the number of records/rows that needs to be examined.

PostgreSQL provides the index methods B-tree, hash, GiST, and GIN. By default, the CREATE INDEX command creates B-tree indexes, which works well with all the comparison operators, or with operators such as BETWEEN, IN, IS NULL, LIKE etc.
Hash indexes can only handle equality comparisons.

More about Hash Indexes:

The reason hash indexes are used is because hash tables are extremely efficient when it comes to just looking up values. So, queries that compare for equality to a string work well. For instance, the query where we fetch employees whose name is 'RAM' could benefit from a hash index created on the Name column.

The way a hash index would work is that the column value will be the key into the hash table and the actual value mapped to that key would just be a pointer to the row data in the table. Since a hash table is basically an associative array, a typical entry would look something like “RAM => 0×28939″, where 0×28939 is a reference to the table row where RAM is stored in memory. Looking up a value like “RAM” in a hash table index and getting back a reference to the row in memory is obviously a lot faster than scanning the table to find all the rows with a value of “RAM” in the Name column.

Reason for choosing this topic:

While going through the source code and the mailing lists, i came across the following lines

“Note: Testing has shown PostgreSQL's hash indexes to perform no better than B-tree indexes, and the index size and build time for hash indexes is much worse. For these reasons, hash index use is presently discouraged.”

While it's evident that Hash index operations are not currently WAL logged and also that it works only for statements that check for equality, hence the usage being discouraged. However, I decided to check for myself how it compares against B-Tree indexes in terms of speed.

Here, I try to present the results of my comparison between the two indexing techniques.

Creation of Tables, Comparison:

First of all, initialize a schema test (As was shown in the postgres_installation).

Enter the schema

```
./bin/psql test
```

Now create a test_table with random_string as a column.

```
Create table test_table (random_string text);
```

To test the index creation time, we would need atleast a million rows. Let's generate that using a perl script. (insertdata.pl)

The script is as follows:

```
#!/usr/bin/perl -w
use strict;
print generate_random_string() . "\n" for 1 .. 10_000_000;
exit;
sub generate_random_string {
    my $word_count = 2 + int(rand() * 4);
    return join(' ', map { generate_random_word() } 1..$word_count);
}
```

```
  sub generate_random_word {
    my $len = 3 + int(rand() * 5);
    my @chars = ( "a".."z", "A".."Z", "0".."9" );
    my @word_chars = map { $chars[rand @chars] } 1.. $len;
    return join '', @word_chars;
}
```

The above script generates a million random strings each of which has a few words. The output of this file has to be redirected to a .lst file. That is dont by using

```
Perl insertdata.pl > data.lst
```

This will redirect the output to the data.lst file.

Next step is to import the data from the data.lst file to the test_table relation.

That is done by using:

```
test=#\copy test_table from 'data.lst';
```

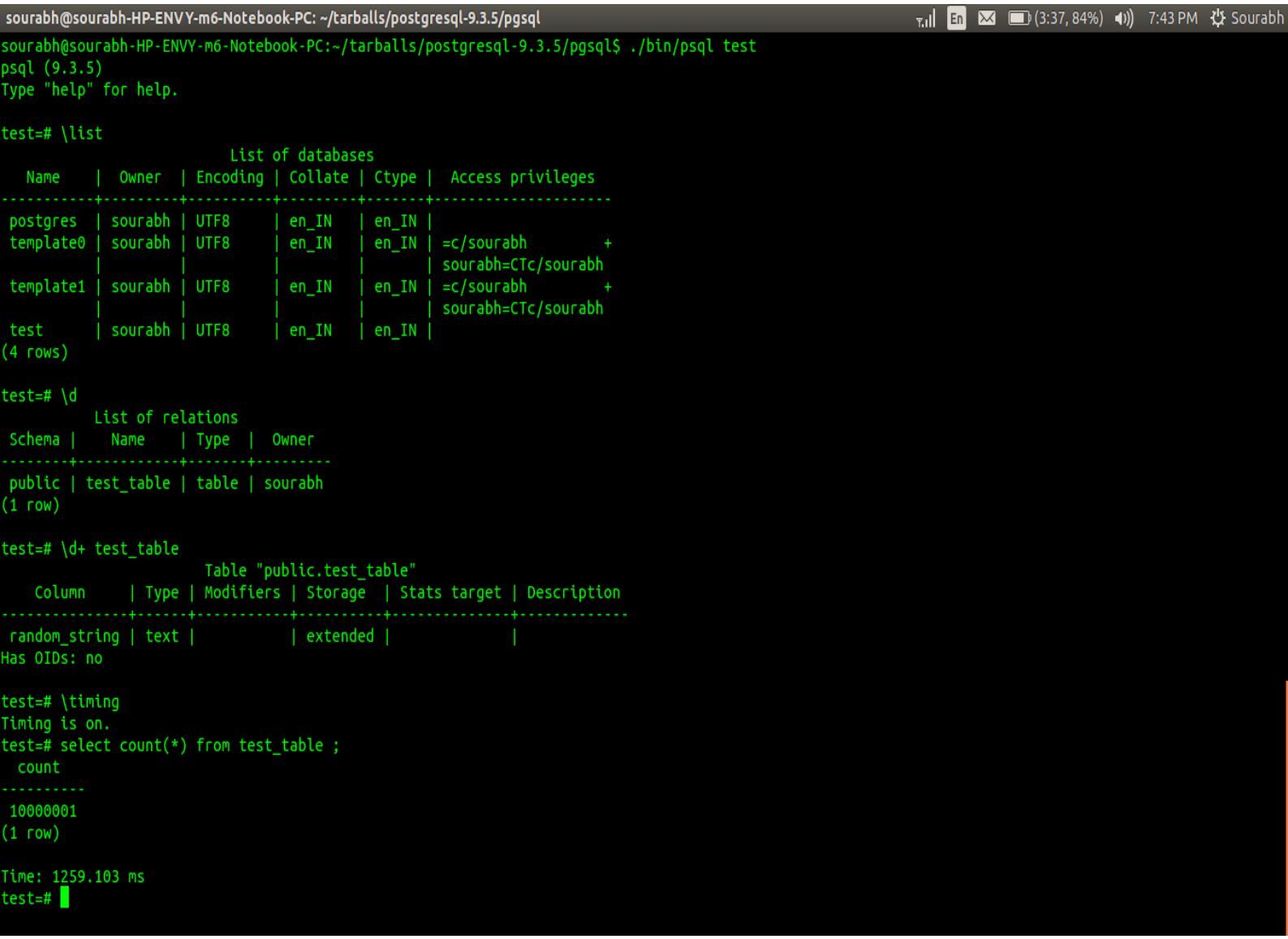
Now the table test_table has 10M rows.

We can check this by using

Select count(*) from test_table;

Also, \timing can be used to turn on the timing.

The results until this point are as seen in the following snapshot:



Now the next step is to check the speed for index creation.

For this, i will need to create the following indexes:

- Hash Index
- B-Tree Index
- Unique B-Tree Index
- Unique Hash Index

The command to create an index for a column is:

CREATE [UNIQUE] INDEX [index-name] ON table [USING method](column-name(s));

where,
index name can be anything
unique hash is not implememted
method can either be hash or btree
column names are the column(s) on which the indexes are to be created.

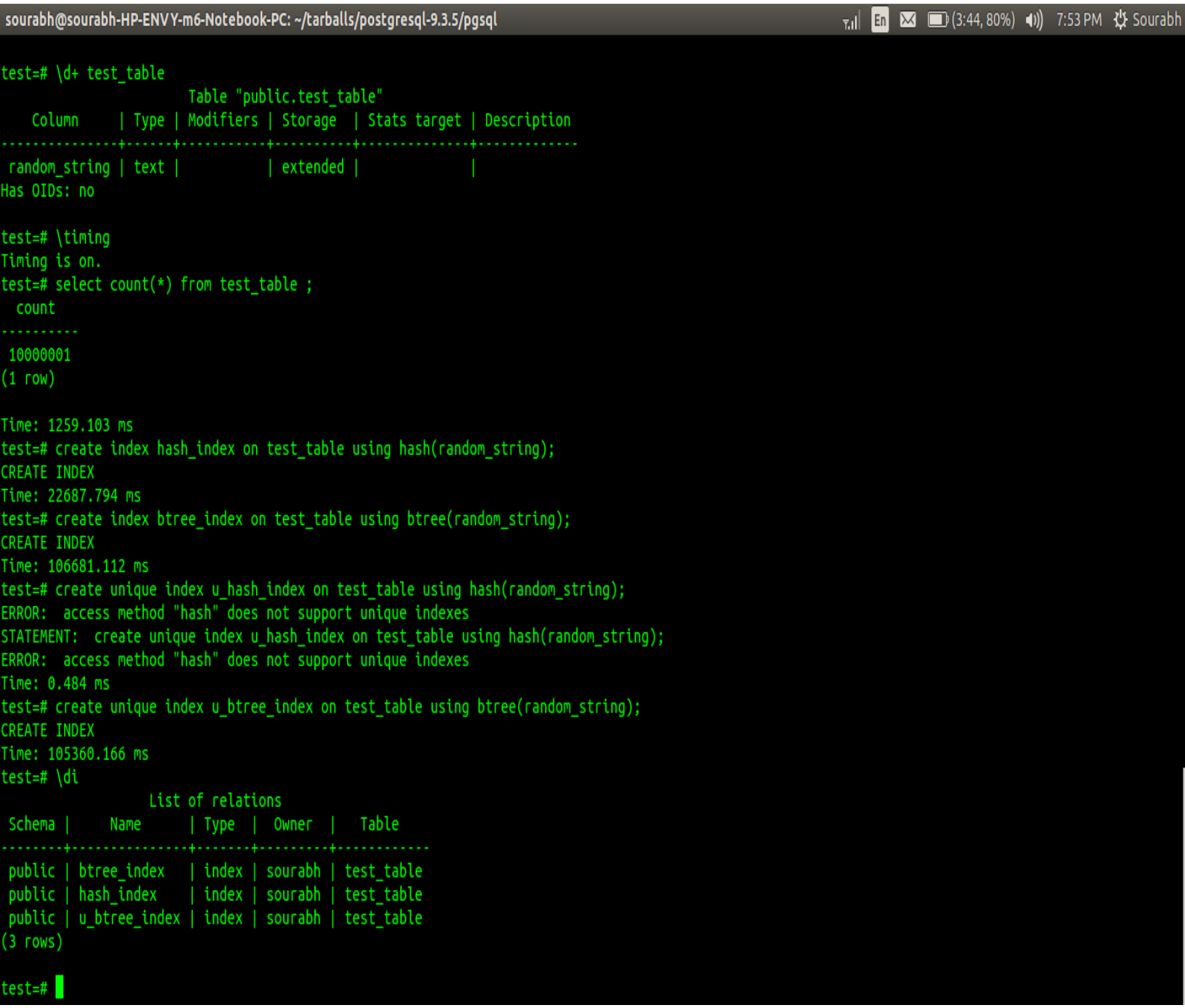
The names given are as follows:

btree_index for B-Tree

u_btree_index for Unique B-Tree

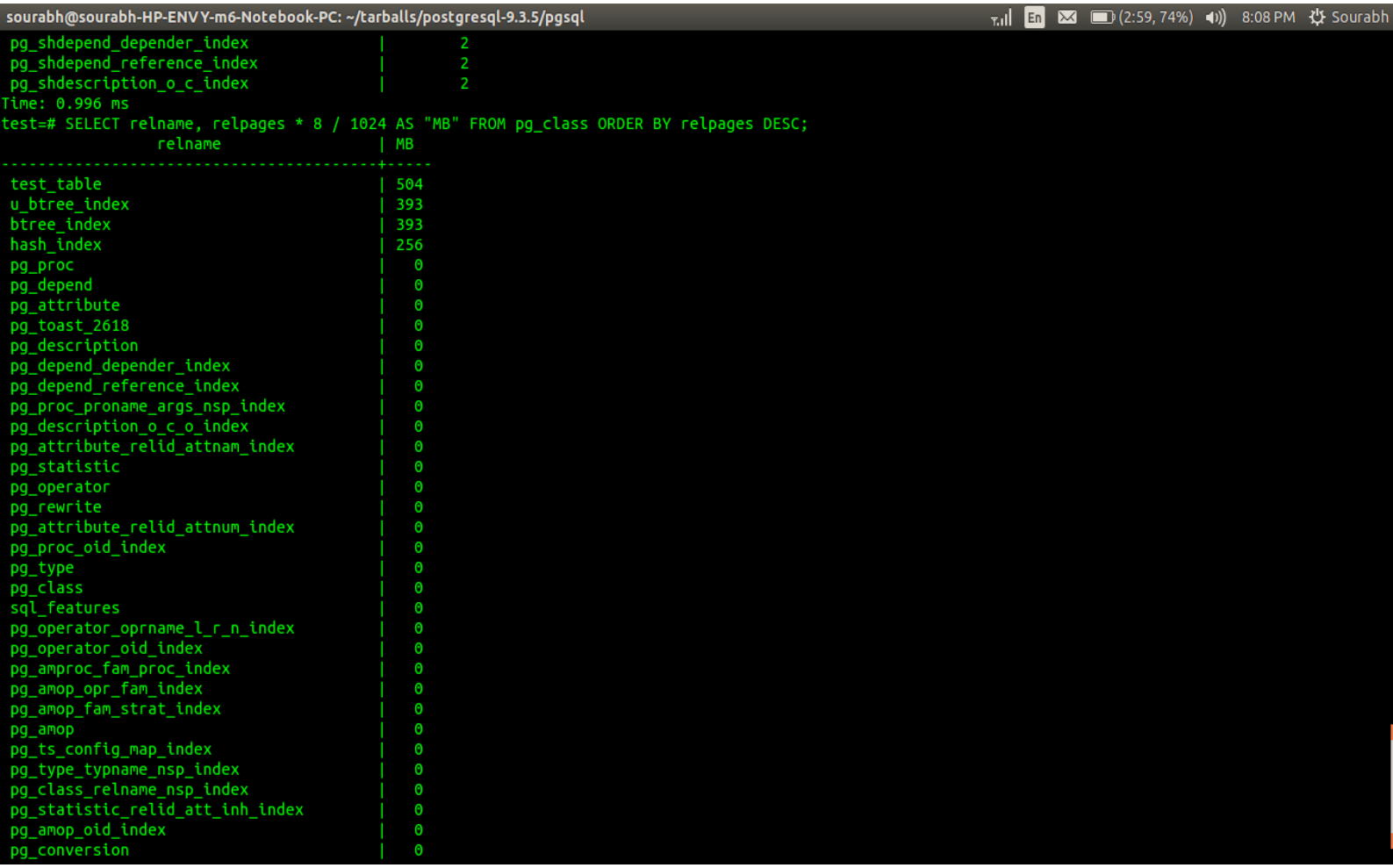
hash_index for Hash

The snapshot of the Database at that time is as follows:



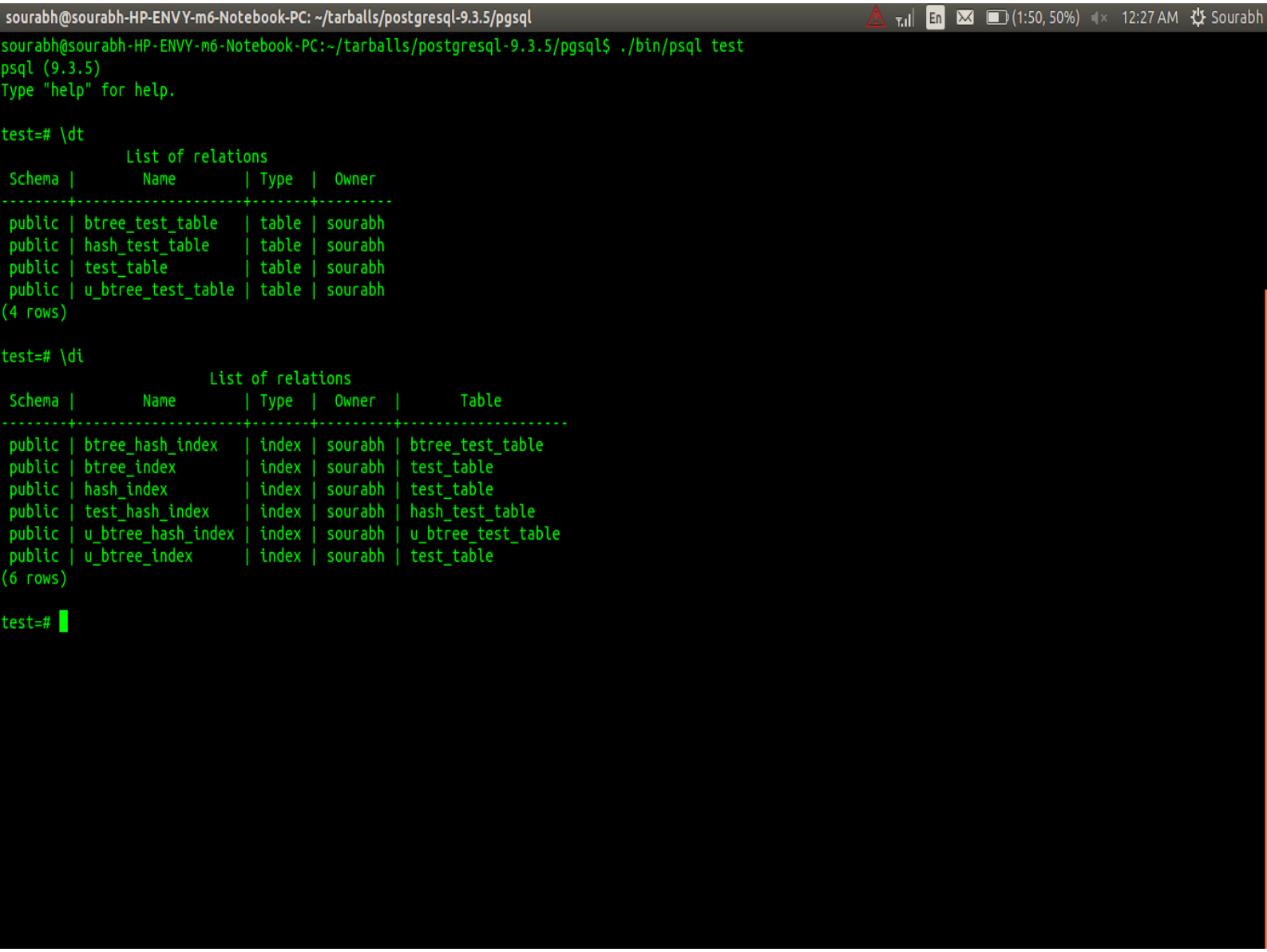
As we can observe from the above snapshot, the index creation time for hash index is much lesser than the time taken to create a B-Tree index. This was observed to be true even when repeated again after dropping the index.

What about the size of the indexes created? As per what we checked, the size of the Hash index was much smaller than the size of the B-tree index. Here's a snapshot comparing the size in MB.



How do they fare against each other when it comes to statements that test for equality?
We try this out for 2 types of queries. One having values present in the table and one which doesn't.

To enable simultaneous execution of queries, I copied the same table thrice and renamed them and created respective indexes as shown below.



Now type the following perl script and run it (compare.pl).

```
#!/usr/bin/perl -w
use strict;
use Benchmark qw( cmpthese );
use DBI;

my @existing_strings = (
    'PDVfHsH 2zm 3Lls ncdh Xf006',
    'VfR5L HIVWFt P2H5 yWxC6',
    'hyh ru6z19',
    'nJkC pU4V7 l4f 1ewgmxd 6BN',
    '6Mmn D1NeP IMOT2j',
    '53xWE6g fop0OM moBi zOP',
    '2EgFHwr AwpD9tt 5GY zewDok OLIE',
    '4YAQAF B87GBC7 GtIPEBC lgML awGO',
    'uxJKdN dQela AKC OIyqiTL X1qovd',
    'uzEFodF oFdH b9zHQ HWYQp1p',
);

my @nonexisting_strings = (
    '3zwy PJe 314P',
    'pqo6d n6j 5t1V',
    'XEyoX ciKRh01d iJrSS',
    'pa3r29P V16Gv1gy ayEv',
    'TB0f9 5TE wbP1GI',
    'T0btpG aK6R1hF x1mXlo',
    '2T2 xdMf6k1b WUuZi',
```

```
    'dPV xipP4 az1F',
    't3IC Qjbtx5V H186',
    'XLyiZa6 Wj1 XK1Kz',
);

my $dbh = DBI->connect(
    'dbi:Pg:dbname=test;host=127.0.0.1;port=5432', "sourabh", "",
    {
        'AutoCommit' => 1,
        'RaiseError' => 1,
        'PrintError' => 1,
    }
);

print "Testing existing keys:\n";
cmpthese(
    50000,
    {
        'existing_hash'            => sub
{ fetch_from( 'hash_test_table',         \@existing_strings ); },
        'existing_btree'           => sub
{ fetch_from( 'btree_test_table',        \@existing_strings ); },
        'existing_unique_btree'    => sub
{ fetch_from( 'u_btree_test_table', \@existing_strings ); },
    }
);

print "\nTesting nonexisting keys:\n";
```

```
cmpthese(
    50000,
    {
        'nonexisting_hash'         => sub { fetch_from( 'hash_test_table',
\@nonexisting_strings ); },
        'nonexisting_btree'        => sub
{ fetch_from( 'btree_test_table',        \@nonexisting_strings ); },
        'nonexisting_unique_btree' => sub
{ fetch_from( 'u_btree_test_table', \@nonexisting_strings ); },
    }
);

exit;

sub fetch_from {
    my ( $table, $values ) = @_;
    my $sql = 'SELECT * FROM ' . $table . ' WHERE random_string =
ANY( ? )';
    my $rows = $dbh->selectall_arrayref( $sql, undef, $values );
    return;
}
```

This creates a benchmark comparison between the three indexes. The results are as shown in the snapshot that follows.

```
sourabh@sourabh-HP-ENVY-m6-Notebook-PC:~/tarballs/postgresql-9.3.5/pgsql$ perl ../compare.pl
Testing existing keys:
                      Rate existing_unique_btree  existing_hash existing_btree
existing_unique_btree 3891/s                   --           -1%           -9%
existing_hash         3934/s                   1%            --           -8%
existing_btree        4277/s                  10%            9%            --

Testing nonexisting keys:
                         Rate nonexisting_unique_btree nonexisting_btree nonexisting_hash
nonexisting_unique_btree 4822/s                     --               -1%              -4%
nonexisting_btree        4888/s                     1%                --              -3%
nonexisting_hash         5015/s                     4%                3%               --
sourabh@sourabh-HP-ENVY-m6-Notebook-PC:~/tarballs/postgresql-9.3.5/pgsql$
```

Conclusion:

In my Comparison, i found out that Hash index creation need not always be slower as compared to B-Tree index creation. Also, in the test where we check for non existing strings, Hash index is 4%-6% faster than B-Tree index. While this may be true, it is still a marginal difference and also does not justify the usage of hash indexes over B-Tree indexes which can handle much more types of Queries.