Performance Tuning and Optimization in Postgres

The goal of this assignment is to explore performance tuning of a relatively large database in the Postgres database system. The database was created from a dump of the Stackoverflow website (see the last page of this assignment for details on the schema).

Your goal is to evaluate the performance of three queries under different configurations of the database system. For the purposes of this assignment, we will consider a database configuration to be the set of indexes present in the system along with a possible clustering of each table according to one of its indexes. The Default configuration is one which has no indexes, and this is the state of your database when you start the provided machine image in Amazon Web Services.

Basic tools for this assignment

To complete the assignment, you will need to (1) create and modify indexes; (2) inspect query plans; (3) monitor page I/Os. Each of these are described below:

(1) Creating and Modifying Indexes
Indexes are critical to performance in databases. You will need to build one or more indexes, and you will optionally choose to cluster a table according to an index. The following commands are used to manage indexes:

- CREATE INDEX [name of index] ON [table name] USING btree ([attribute-list]);
- CREATE INDEX [name of index] ON [table name] USING hash ([attribute-list]);
- CLUSTER [tablename] USING [indexname];
- ANALYZE;
- DROP INDEX [name of index];
- In psql, the meta-command \di will list all existing indexes.

The above commands are used to explicitly create or remove indexes. Indexes may also be created automatically by the system. In particular, when a key constraint is defined for a table, Postgres will automatically create an index on the key attributes. In the default configuration, no key constraints are specified (and no indexes are present). But this command will result in a new index:

ALTER TABLE [tableName] ADD PRIMARY KEY( [attribute] );
In addition, recall that when a table is clustered it is physically reordered according to the index attributes. A table can be clustered according to only one index. After creating an index or clustering a table, it is very important to run ANALYZE which updates the database statistics. Otherwise, the optimizer may make poor choices in its use of the index.

(2) Inspecting Query Plans
To see the plan chosen by the optimizer for any SELECT statement, you can add EXPLAIN immediately before the SELECT. The query will not be executed. Instead, a textual description of the plan will be output.

(3) Monitoring Page I/Os
Postgres keeps statistics about the number of page I/Os performed by the system. You can access these statistics by running simple queries:

SELECT * FROM pg_statio_user_tables;

This query returns each table in the system along with I/O statistics. The most important values for this assignment are heap_blks_read and idx_blks_read. Note that idx_blks_read reports the number of pages read from all indexes related to the given table. The following query reports statistics for each index in the system (although this command is not strictly required for this assignment):

SELECT * FROM pg_statio_user_indexes;

The statistics collected are cumulative, so that each time you run a query, the number of blocks read in the execution of that query will be added to the current statistics. Therefore, if you wish to count statistics for an individual operation, it’s often convenient to reset the statistics by running this query:

SELECT pg_stat_reset();
Assigned Tasks

For each of the three queries listed on the next page, and each configuration described below, evaluate the query in the configuration by doing the following:

(a) Run EXPLAIN on the query and describe the query plan informally. (Write about one sentence describing the access paths, join algorithms, etc.)
(b) Compute and report I/O statistics for the query (Include heap_blks_read and idx_blks_read for each table involved in the query and report the total of the blocks read.)
(c) Compute the real time to execute the query (Compute an average over a few runs.)

Begin in the default configuration:

1) Evaluate Query1 in the default configuration.
2) Evaluate Query2 in the default configuration.
3) Evaluate Query3 in the default configuration.

Change to Configuration1 by creating these constraints:

ALTER TABLE posts ADD PRIMARY KEY(id);
ALTER TABLE votes ADD PRIMARY KEY(id);
ALTER TABLE users ADD PRIMARY KEY(id);
ALTER TABLE badges ADD PRIMARY KEY(id);
ALTER TABLE comments ADD PRIMARY KEY(id);
ALTER TABLE votes ADD CONSTRAINT votes_fkey FOREIGN KEY (userid) REFERENCES users(id) MATCH FULL;
ANALYZE;

4) Evaluate Query1 in Configuration1.
5) Evaluate Query2 in Configuration1.
6) Evaluate Query3 in Configuration1.

Change to Configuration2 by creating an btree index on posts.creationdate and clustering posts on this index.

CREATE INDEX posts_creationdate ON posts USING btree(creationdate);
CLUSTER posts USING posts_creationdate;
ANALYZE;

7) Evaluate Query1 in Configuration2.
8) Evaluate Query2 in Configuration2.
9) Evaluate Query3 in Configuration2.

(Extra Credit)
10) Find an alternative configuration (i.e. additional indexes and/or clustering) for which the total I/O cost of running all three queries is lower than that of Configuration2.
Example Queries

These queries are also available to download from the Assignments page.

**Query 1**
```
select
  owneruserid, 
  avg(score) as avg_score,
  sum(viewcount) as total_views,
  min(creationdate) as first_active_time,
  max(creationdate) as last_active_time,
  count(*) as total_postings
from
  posts
where
  creationdate<='2010-03-01' and
  creationdate>='2010-02-01'
group by
  owneruserid
order by
  owneruserid
limit 10;
```

**Query 2**
```
select
  p.id, 
  p.answercount, 
  p.commentcount, 
  avg(c.score)
from
  posts p, comments c
where
  p.id = c.postid
  AND p.posttypeid='1'
  AND c.score is not null
  AND p.creationdate >='2008-10-01'
  AND p.creationdate<=2008-11-01'
group by
  p.commentcount,p.answercount,p.id
order by
  p.commentcount desc, 
  p.answercount desc
limit 10;
```

**Query 3**
```
select
  p.id, 
  p.creationdate, 
  u.displayname, 
  v.votetypeid, 
  p2.score
from
  posts p, posts p2, votes v, users u
where
  p.creationdate>=2008-10-01'
  AND p.creationdate='2008-11-01'
  AND p.posttypeid='1'
  AND p.favoritecount>0
  AND p2.commentcount>1
  AND v.votetypeid='5'
  AND p.id=v.postid
  AND u.id=v.userid
  AND p.id=p2.parentid
order by
  p.creationdate
limit 10;
```
Schema for StackOverflow database

badges(id integer, userid integer, name varchar(256), date timestamp);

comments(id integer, postid integer, score integer, text text, creationdate timestamp, userid integer);

posts(id integer, posttypeid integer, parentid integer, acceptedanswerid integer, creationdate timestamp, score integer, viewcount integer, body text, owneruserid integer, lasteditoruserid integer, lasteditordisplayname varchar(256), lasteditdate timestamp, lastactivitydate timestamp, communityowneddate timestamp, closeddate timestamp, title text, tags text, answercount integer, commentcount integer, favoritecount integer);

tags(postid integer, name varchar(256))

users(id integer, reputation integer, creationdate timestamp, displayname varchar(256), emailhash text, lastaccessdate timestamp, websiteurl text, location text, age integer, aboutme text, views integer, upvotes integer, downvotes integer);

votes(id integer, postid integer, votetypeid integer, userid integer, creationdate timestamp, bountyamount integer);

Statistics about the StackOverflow database

<table>
<thead>
<tr>
<th>Table Name</th>
<th>Tuples</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>badges</td>
<td>825377</td>
<td>5792</td>
</tr>
<tr>
<td>comments</td>
<td>2990470</td>
<td>76545</td>
</tr>
<tr>
<td>posts</td>
<td>3059105</td>
<td>265624</td>
</tr>
<tr>
<td>tags</td>
<td>2490536</td>
<td>13460</td>
</tr>
<tr>
<td>users</td>
<td>301260</td>
<td>5505</td>
</tr>
<tr>
<td>votes</td>
<td>7831298</td>
<td>49919</td>
</tr>
</tbody>
</table>