Amee Sankhesara
Vinaya D Bhat

CS 487/587 Database Implementation
Spring 2019
Database Benchmarking Project - Part II

Which option / system (s) you will be working with and why you chose it (them) (20 pts)

We choose the 2nd option: Evaluate the Postgres system with different parameter values and optimizer options. We know that PostgreSQL is a comprehensive, sophisticated database system offering countless features. In Postgres system the user defines what should be done through SQL queries but does not specify how the queries should be executed. There are many ways in which SQL queries can be processed. The query optimizer attempts to determine the most efficient way to execute a query based on their estimated costs and creates an execution plan. This plan is later executed step-by-step by the database. We choose this option to analyse the default query plans chosen by the Postgres system and also wanted to observe how the system behaves when this default query plan is not made available to it (evaluating the other query plans the system chooses). We also wanted to learn how we can increase performance of query by modifying existing default configuration parameters. Another reason for choosing this option was to see how Postgres chooses query plans based on the selectivity. We know that Postgres does advanced cost-based query optimization, so we also wanted to compare the query costs of different query plans (each doing same computation). We also wanted to observe how the memory configurations affects the execution time of a query in PostgreSQL.

System Research :

We are going to do research on postgres parameters.
We will do research on below postgres parameters.

- **Enable_nestloop** : By setting this parameter on/off we can enable/disable use of nested loop join in query plan. By default value of this parameter is On(Enabled). It is not possible to suppress use of nested loop join in query plan. But by setting this parameter's value off discourages postgres to use nested loop join in query plan if any other option is available. It will check other options and choose one of them in query plan.

- **Enable_mergejoin** : By setting this parameter on/off we can enable/disable use of merge join in query plan. By default value of this parameter is On(Enabled). If you set this parameter off then it will avoid use of merge join in query and it will use one of other available join option in query plan.

- **Enable_seqscan** : By setting this parameter on/off we can enable/disable use of sequential scan in query plan. By default value of this parameter is On(Enabled). It is not possible to suppress use of sequential scan in query plan. But by setting this parameter's value off discourages postgres to use sequential scan in query plan if any other option is available. It will check other options and choose one of them in query plan.

- **Enable_indexscan** : By setting setting this parameter on/off we can enable/disable use of index scan in query plan. By default value of this parameter is On(Enabled).

- **Enable_hashagg** : By setting setting this parameter on/off we can enable/disable use of hashed aggregation in query plan. By default value of this parameter is On(Enabled). If you set this parameter off then it will avoid use of hashed aggregation in query and it will use one of other available option like group aggregation in query plan.

- **Enable_sort** : By setting this parameter on/off we can enable/disable use of explicit sort steps in query plan. By default value of this parameter is On(Enabled). It is not possible to suppress use of explicit sort steps in query plan. But by setting this parameter's value off discourages postgres to use explicit sort steps in query plan if any other option is available. It will check other options and choose one of them in query plan.

- **Work_mem** : This parameter specifies the amount of memory to be used by internal sort operations and hash tables before writing to temporary disk files. If a lot of complex sorts are happening, and you have enough memory, then increasing the work_mem parameter allows PostgreSQL to do larger in-memory sorts which will be faster than disk-based equivalents.

  Note that for a complex query, many sort or hash operations might be running in parallel. Each operation will be allowed to use as much memory as this value specifies before it starts to write data into the temporary files. There is one possibility that several sessions could be doing such operations concurrently. Therefore, the total memory used could be many times the value of work_mem parameter. it is necessary to keep this fact in mind when choosing the value. Sort operations are used for ORDER BY, DISTINCT and merge joins. Hash tables are used in hash joins, hash-based processing of IN subqueries and hash based aggregation.

  The default value of work_mem = 4MB.

Performance Experiment Design:

Performance Experiment - 1:

- In this performance experiment we are going to test performance of different join algorithms.
- For this experiment we have created two tables fifteenhundredktup and thousandktup in PostgreSQL database using data generation code of wisconsin benchmark which we have implemented in project part-1. Table fifteenhundredktup has 1500000 tuples and table thousandktup has 1000000 tuples.

  Below are the queries:

- Query - 1 : Below query will retrieve rows from table fifteenhundredktup for which unique1 values exists in table thousandktup.

  select * from fifteenhundredktup
  where unique1 in (select unique1 from thousandktup)

  Expected Result: According to us it will give nested loop join in query plan. Because both table having large number of rows and it has more than 10% selectivity. Both tables thousandktup and fifteenhundredktup will be scanned sequiential. So it will give nested loop join in query plan.

- Query - 2: Query-1 with setting parameter Enable_nestloop to off.

  Set Enable_nestloop = Off

  select * from fifteenhundredktup
  where unique1 in (select unique1 from thousandktup)

  Expected Result : According to us when we will turn off Enable_nestloop parameter then it will use merge join in query plan. Because when we turn off this parameter it will have other option of hash and merge join only. Here the both tables are very large so it will not fit in memory. So according to us it will use merge join in query plan. And it will improve performance of query.

- Query - 3: Query-1 with setting parameter Enable_mergejoin to off.

  Set Enable_mergejoin = Off

  select * from fifteenhundredktup
  where unique1 in (select unique1 from thousandktup)

Expected Result : According to us when we will turn off Enable_mergejoin parameter then it will force to use hash join in query plan. Here the both tables are very large so it will divide data into buckets and will perform hash join. And it will make performance worse than merge join as tables are very large and not fitting in memory.

Performance Experiment - 2:

- In this performance experiment we are going to test performance of index scan and sequential scan with selectivity.
- For this experiment we have created table fifteenhundredktup in PostgreSQL database using data generation code of wisconsin benchmark which we have implemented in project part-1. Table fifteenhundredktup has 1500000 tuples.

  Below are the queries:

- Query - 1: Below query will retrive data from fifteenhundredktup having unique2 between 0 and 120000 and stringu1 having pattern that starting 3 character is A.

  select * from fifteenhundredktup where stringu1 like 'AAA%' and unique2 between 0 and 120000

  Expected Result: Here we are fetching 0 to 120000 number of records from total 1500000 number of records. So, selectivity is less than 10%. Unique2 is primary key of table. So it has clustered index. And selectivity is less than 10% so It will use Index scan for this query according to us. First it will fetch data which has unique2 between 0 and 120000 and then on that data it will apply filter on column stringu1.

- Query - 2: Below query will retrive data from fifteenhundredktup having unique2 greater than 150000 and stringu1 having pattern that starting 3 character is A.

  select * from fifteenhundredktup where stringu1 like 'AAA%' and unique2 > 150000

  Expected Result : Here we are fetching more than 10% records so selectivity is greater than 10%. So it will use sequential scan for this query even though unique2 has clustered index.

- Query - 3: Query-1 with setting parameter Enable_indexscan to Off.

  Set Enable_indexscan = off

  select * from fifteenhundredktup where stringu1 like 'AAA%' and unique2 between 0 and 120000

Expected Result : According to us when we set parameter Enable_indexscan to Off it will use Bitmap Heap Scan on table. Because selectivity is less than 10% so for sequential scan there is not enough data. So it will use Bitmap heap scan on both table. First it will find portion of data according to where clause then it will perform join operation on the fly and then it will get result from it.

- Query - 4: Query-2 with setting parameter Enable_seqscan to Off.

  Set Enable_seqscan = off

  select * from fifteenhundredktup where stringu1 like 'AAA%' and unique2 > 150000

  Expected Result : According to us when we set parameter Enable_seqscan to Off. It will force to use index scan even though its selectivity is greater than 10%. And it will give worse performance than sequential scan.

Performance Experiment - 3:

- In this performance experiment we are going to test performance of different types of aggregations.
- For this experiment we have created one table thousandktup in PostgreSQL database using data generation code of wisconsin bench mark which we have implemented in project part-1. Table thousandktup has 1000000 tuples.

  Below are the queries:

- Query1 : Below query will give minimum value of evenonepercent for values of oddonepercent.

  set Enable_sort = off
  set Enable_hashagg = on

  select oddonepercent,min(evenonepercent) as minevenonepercent
  from thousandktup
  group by oddonepercent

  Expected Result : According to us it will use hash aggregate in query. Because this query is using aggregate function and also we have disabled explicit sort steps here. So it will fit into memory and it will use hash aggregate. And according to us it will perform better than group aggregate having explicit sort enabled.

- Query2: Query-1 with setting parameter Enable_sort = on and Enable_hashagg = off.
  set Enable_sort = on

set Enable_hashagg = off

select oddonepercent,min(evenonepercent) as minevenonepercent
from thousandktup
group by oddonepercent

Expected Result : According to us it will use group aggregate in query. There is a aggregate function in query and we have enabled explicit sort steps here. So it will not fit into memory and it will use group aggregate. According to us its performance will be worse than hash aggregation having disabled explicit sort steps.

Performance Experiment - 4 :

- In this performance experiment we are going to test performance of memory management and execution time using work_mem parameter.
- For this experiment we have created one table thousandktup1 in PostgreSQL database using data generation code of wisconsin bench mark which we have implemented in project part-1. Table thousandktup1 has 1000000 tuples.

  Below are the queries:

- Query - 1 : This query helps to find those values of "unique1" column whose value is greater than 1000 and respective unique2 is greater than 1000 and tenpercent value is greater than 10000.

  set work_mem = '128MB'

  select * from (
  select * from (select * from thousandktup1
  where tenpercent > 10000) t where t.unique2 > 1000) s where s.unique1 > 1000

  Expected Result : For the initial run, default value of work_mem = 4MB and when modified to 128MB, time consumption should decrease exponentially.

<u>Lessons Learned: Include lessons learned or issues encountered (20 pts)</u>

During this phase 2 of the project we did get to learn a lot about the Postgres parameters and optimization options. We were able to see how the selectivity plays a major role in query plan selection. We were able to see the effect of data size on query optimizer. It taught us a great deal about various parameters that can be changed in Postgres. It gave us a in depth knowledge of how configuration parameters can affect the behavior of database systems. Before doing this research our understanding of Postgres was limited to execution of queries without thinking what the underlying query optimizer actually does. Whenever

we executed a query, we knew that Postgres would end up choosing the most cost-effective query plans but we had no experience in changing the parameters so that postgres query optimizer could not choose certain plans while evaluating query. We have also learnt how we can get better query performance by tweaking configuration parameters like work_mem. We were able to learn alot about configuration parameters which was very new to us. Initially we found it difficult to think of queries and come up with various diverse experiments. Sometimes on execution of a query the system can cache the result , so when we execute the same query next time we do not know whether the system is returning the cached result or actually taking the data from disk. This was one of our main concerns since cache result gives better performance.