Innovative Assignment 2

Gahan Saraiya (18MCEC10), Rushi Trivedi (18MCEC08), Raj Kothari (18MCEC07)

18mcec10@nirmauni.ac.in, 18mcec08@nirmauni.ac.in, 18mcec07@nirmauni.ac.in

I. Introduction

Aim of this assignment is to analyze impact of various indexes on any modern database (SQL or NoSQL).

Steps to be followed:

- Select Database
- Analyze performance and cost impact of indexes for various type of query

I. Hash Table inside PostgreSQL

Postgres[1] scans over all the records in one of the tables from the join and saves them in a hash table.

II. Implementation

I. Considerations

Variable	Value	Detail
Database	PostgreSQL	
Analyze scale factor	0.1	Number of tuple inserts, updates, or deletes prior to analyze as a fraction of reltuples.
Analyze Threshold	50	Minimum number of tuple inserts, updates, or deletes prior to analyze.
Block Size	8192	Shows the size of a disk block.
CPU Index Tuple cost	0.005	Sets the planner's estimate of the cost of processing each index entry during an index scan.
CPU operator cost	0.0025	Sets the planner's estimate of the cost of processing each operator or function call.

[1] referred as acronym to PostgreSQL

CPU tuple cost	0.01	Sets the planner's estimate of the cost of processing each tuple (row).
cursor tuple fraction	0.1	Sets the planner's estimate of the fraction of a cursor's rows that will be retrieved.
deadlock timeout	1s	Sets the time to wait on a lock before checking for deadlock.
commit delay	0μs	Sets the delay in microseconds between transaction commit and flushing WAL[2] to disk.
commit sibling	5μs	Sets the minimum concurrent open transactions before performing commit_delay.
Effective cache size	4GB	Sets the planner's assumption about the size of the disk cache.
Effective IO concurrency	1	Number of simultaneous requests that can be handled efficiently by the disk subsystem.
Log Rotation Age	1d	Automatic log file rotation will occur after N minutes.
Log Rotation Size	10MB	Automatic log file rotation will occur after N kilobytes.
Maintenance Work Memory	64MB	Sets the maximum memory to be used for maintenance operations.
Max Connections	100	Sets the maximum number of concurrent connections.
Max files per process	1000	Sets the maximum number of simultaneously open files for each server process.
Max function arguments	100	Shows the maximum number of function arguments.
Max Index Keys	32	Shows the maximum number of index keys.
Max Stack depth	2MB	Sets the maximum stack depth, in kilobytes.
Max WAL Size	1GB	Sets the WAL size that triggers a checkpoint.
Min WAL Size	80MB	Sets the minimum size to shrink the WAL to.
Segment Size	1GB	Shows the number of pages per disk file.

[2]WAL - Write Ahead Logging

Work Memory	4*MB*	Sets the maximum memory to be used for query workspaces.

Table 1: Consideration and variables

II. Schema

```
--
-- PostgreSQL database dump
--

-- Dumped from database version 9.5.14
-- Dumped by pg_dump version 10.3

SET statement_timeout = 0;
SET lock_timeout = 0;
SET idle_in_transaction_session_timeout = 0;
SET client_encoding = 'UTF8';
SET standard_conforming_strings = on;
SELECT pg_catalog.set_config('search_path', '', false);
SET check_function_bodies = false;
SET client_min_messages = warning;
SET row_security = off;

SET default_tablespace = '';

SET default_with_oids = false;

--
-- Name: collection_title; Type: TABLE; Schema: public; Owner: postgres
--

CREATE TABLE public.collection_title (
    id integer NOT NULL,
    title_id character varying(16) NOT NULL,
    ordering integer NOT NULL,
    title character varying(512) NOT NULL,
    region character varying(8),
    language character varying(16),
    types character varying(32),
    attributes character varying(256),
```

```
    description text,
    is_original_title boolean NOT NULL
);


ALTER TABLE public.collection_title OWNER TO postgres;

--
-- Name: collection_title_id_seq; Type: SEQUENCE; Schema: public; Owner:
 ↪  postgres
--

CREATE SEQUENCE public.collection_title_id_seq
    START WITH 1
    INCREMENT BY 1
    NO MINVALUE
    NO MAXVALUE
    CACHE 1;


ALTER TABLE public.collection_title_id_seq OWNER TO postgres;

--
-- Name: collection_title_id_seq; Type: SEQUENCE OWNED BY; Schema: public; Owner:
 ↪  postgres
--

ALTER SEQUENCE public.collection_title_id_seq OWNED BY
 ↪  public.collection_title.id;


--
-- Name: collection_title id; Type: DEFAULT; Schema: public; Owner: postgres
--

ALTER TABLE ONLY public.collection_title ALTER COLUMN id SET DEFAULT
 ↪  nextval('public.collection_title_id_seq'::regclass);


--
-- Name: collection_title collection_title_pkey; Type: CONSTRAINT; Schema:
 ↪  public; Owner: postgres
```

```
--

ALTER TABLE ONLY public.collection_title
    ADD CONSTRAINT collection_title_pkey PRIMARY KEY (id);


--
-- PostgreSQL database dump complete
--
```

III. Status

Total Entries 2096169

III. Query Impact

I. Exact Match/ Point Query

```
EXPLAIN ANALYZE select * from collection_title where title like '%Batman%' and
↪  title_id='tt0060153' and region='US';
```

Table 2: Query Plan

Index Scan using collection_title_pkey on collection_title	(cost=0.43..8.45 rows=1 width=107) (actual time=0.035..0.036 rows=1 loops=1)
Index Cond	$(id = 334596)$
Planning time	0.127ms
Execution time	0.078ms

II. Partial Match

```
explain analyze select * from collection_title where ordering=1;
```

Table 3: Query Plan

Seq Scan on collection_title	(cost=0.00..48832.51 rows=859800 width=107) (actual time=0.020..305.430 rows=910215 loops=1)
Filter	$(ordering = 1)$
Rows Removed by Filter	1185954
Total Rows	2096169
Planning time	0.091ms
Execution time	334.993ms

```
explain analyze select id,title_id, title from collection_title where title like
↪  '%avenger%';
```

Table 4: Query Plan

Seq Scan on collection_title	(cost=0.00..48832.51 rows=207 width=34) (actual time=0.621..490.246 rows=41 loops=1)
Filter	$((title) :: text \ \ '\%avenger\%' :: text)$
Rows Removed by Filter	1185954
Total Rows	2096128
Planning time	0.120 ms
Execution time	490.282 ms

III. Range Query

```
explain analyze select * from collection_title where id>3566 and id <25996;
```

Table 5: Query Plan

Seq Scan on collection_title	(cost=0.00..48832.51 rows=859800 width=107) (actual time=0.020..305.430 rows=910215 loops=1)
Filter	$(ordering = 1)$
Rows Removed by Filter	1185954
Total Rows	2096169
Planning time	0.091ms
Execution time	334.993ms

```
explain analyze select * from collection_title where id>3566 and id <25996;
```

Table 6: Query Plan

Index Scan using collection_title_pkey on collection_title	(cost=0.43..1089.81 rows=25864 width=107) (actual time=0.168..44.540 rows=22429 loops=1)
Index Cond	$((id > 3566)AND(id < 25996))$
Planning time	0.0140 ms
Execution time	45.695 ms

IV. EXECUTION PLAN NODES

I. Nodes: stream-like

Some nodes are more or less stream-like. They don't accumulate data from underlying nodes and produce nodes one by one, so they have no chance to allocate too much memory.

II. Nodes: stream-like

Some nodes are more or less stream-like. They don't accumulate data from underlying nodes and produce nodes one by one, so they have no chance to allocate too much memory.

II.1 Examples of such nodes include

- Sequential scan, Index Scan
- Nested Loop and Merge Join

- Append and Merge Append
- Unique (of a sorted input)

Even a single row can be quite large. Maximal size for individual postgres value is around 1GB, so this query requires 5GB:

```
WITH cte_1g as (select repeat('a', 1024*1024*1024 - 100) as a1g)
SELECT *
FROM cte_1g a, cte_1g b, cte_1g c, cte_1g d, cte_1g e;
```

III. Nodes: controlled

Some of the other nodes actively use RAM but control the amount used. They have a fall-back behavior to switch to if they realize they cannot fit *work_mem*.

- Sort node switches from quicksort to sort-on-disk
- CTE and materialize nodes use temporary files if needed
- Group Aggregation with DISTINCT keyword can use temporary files
- Exact Bitmap Scan falls back to Lossy Bitmap Scan
- Hash Join switches to batchwise processing if it encounters more data than expected

IV. Nodes: unsafe

They are Hash Agg, hashed SubPlan and (rarely) Hash Join can use unlimited amount of RAM. Optimizer normally avoids them when it estimates them to process huge sets, but it can easily be wrong.

- Sort node switches from quicksort to sort-on-disk
- CTE and materialize nodes use temporary files if needed
- Group Aggregation with DISTINCT keyword can use temporary files
- Exact Bitmap Scan falls back to Lossy Bitmap Scan
- Hash Join switches to batchwise processing if it encounters more data than expected

make the estimates wrong...

```
CREATE TABLE t (a int, b int);
INSERT INTO t SELECT 0, b from generate_series(1, (10^7)::int) b;
ANALYZE t;
INSERT INTO t SELECT 1, b from generate_series(1, (5*10^5)::int) b;
```

After this, autovacuum won't update stats, as it treats the second insert as small w r. t. the number of rows already present.

```
postgres=# EXPLAIN (ANALYZE, TIMING OFF) SELECT * FROM t WHERE a = 1;
QUERY PLAN
-------------------------------------------------------------------------------
Seq Scan on t (cost=0.00..177712.39 rows=1 width=8) (rows=500000 loops=1)
Filter: (a = 1)
Rows Removed by Filter: 10000000
Planning time: 0.059 ms
Execution time: 769.508 ms
```

Then we run the following query

```
postgres=# EXPLAIN (ANALYZE, TIMING OFF)
postgres-# SELECT * FROM t WHERE b NOT IN (SELECT b FROM t WHERE a = 1);
QUERY PLAN
-------------------------------------------------------------------------------------------
Seq Scan on t (cost=177712.39..355424.78 rows=5250056 width=8) (actual
↪  rows=9500000 loops=1)
Filter: (NOT (hashed SubPlan 1))
Rows Removed by Filter: 1000000
SubPlan 1
-> Seq Scan on t t_1 (cost=0.00..177712.39 rows=1 width=4) (actual rows=500000
↪  loops=1)
Filter: (a = 1)
Rows Removed by Filter: 10000000
Planning time: 0.126 ms
Execution time: 3239.730 ms
and get a half-million set hashed.
```

The backend used 60MB of RAM while *work_mem* was only 4MB.

For a partitioned table it hashes the same condition separately for each partition!

```
postgres=# EXPLAIN SELECT * FROM t WHERE b NOT IN (SELECT b FROM t1 WHERE a =
↪  1);
QUERY PLAN
-------------------------------------------------------------------------
Append (cost=135449.03..1354758.02 rows=3567432 width=8)
-> Seq Scan on t (cost=135449.03..135449.03 rows=1 width=8)
Filter: (NOT (hashed SubPlan 1))
SubPlan 1
-> Seq Scan on t1 t1_1 (cost=0.00..135449.03 rows=1 width=4)
Filter: (a = 1)
-> Seq Scan on t2 (cost=135449.03..135487.28 rows=1130 width=8)
```

```
Filter: (NOT (hashed SubPlan 1))
-> Seq Scan on t3 (cost=135449.03..135487.28 rows=1130 width=8)
Filter: (NOT (hashed SubPlan 1))
-> Seq Scan on t4 (cost=135449.03..135487.28 rows=1130 width=8)
Filter: (NOT (hashed SubPlan 1))
-> Seq Scan on t5 (cost=135449.03..135487.28 rows=1130 width=8)
Filter: (NOT (hashed SubPlan 1))
-> Seq Scan on t6 (cost=135449.03..135487.28 rows=1130 width=8)
Filter: (NOT (hashed SubPlan 1))
-> Seq Scan on t7 (cost=135449.03..135487.28 rows=1130 width=8)
Filter: (NOT (hashed SubPlan 1))
-> Seq Scan on t8 (cost=135449.03..135487.28 rows=1130 width=8)
Filter: (NOT (hashed SubPlan 1))
-> Seq Scan on t1 (cost=135449.03..270898.05 rows=3559521 width=8)
Filter: (NOT (hashed SubPlan 1))
```

V. Unsafe nodes: Hash Join

Hash Joins can use more memory than expected if there are many collisions on the hashed side:

```
postgres=# explain (analyze, costs off)
postgres-# select * from t t1 join t t2 on t1.b = t2.b where t1.a = 1;
QUERY PLAN
-------------------------------------------------------------------------------------------
Hash Join (actual time=873.321..4223.080 rows=1000000 loops=1)
Hash Cond: (t2.b = t1.b)
-> Seq Scan on t t2 (actual time=0.048..755.195 rows=10500000 loops=1)
-> Hash (actual time=873.163..873.163 rows=500000 loops=1)
Buckets: 131072 (originally 1024) Batches: 8 (originally 1) Memory Usage: 3465kB
-> Seq Scan on t t1 (actual time=748.700..803.665 rows=500000 loops=1)
Filter: (a = 1)
Rows Removed by Filter: 10000000
postgres=# explain (analyze, costs off)
postgres-# select * from t t1 join t t2 on t1.b % 1 = t2.b where t1.a = 1;
QUERY PLAN
-------------------------------------------------------------------------------------------
Hash Join (actual time=3542.413..3542.413 rows=0 loops=1)
Hash Cond: (t2.b = (t1.b % 1))
-> Seq Scan on t t2 (actual time=0.053..732.095 rows=10500000 loops=1)
-> Hash (actual time=888.131..888.131 rows=500000 loops=1)
Buckets: 131072 (originally 1024) Batches: 2 (originally 1) Memory Usage:
↪ 19532kB
```

```
-> Seq Scan on t t1 (actual time=753.244..812.959 rows=500000 loops=1)
Filter: (a = 1)
Rows Removed by Filter: 10000000
```

V. Measuring RAM usage...

We have to look into /proc filesystem, namely /proc/pid/smaps /proc/7194/smaps comprises a few sections like below which is a private memory segment . . .

```
....
0135f000-0a0bf000 rw-p 00000000 00:00 0
[heap]
Size: 144768 kB
Rss: 136180 kB
Pss: 136180 kB
Shared_Clean: 0 kB
Shared_Dirty: 0 kB
Private_Clean: 0 kB
Private_Dirty: 136180 kB
Referenced: 114936 kB
Anonymous: 136180 kB
AnonHugePages: 2048 kB
Swap: 0 kB
KernelPageSize: 4 kB
MMUPageSize: 4 kB
Locked: 0 kB
VmFlags: rd wr mr mw me ac sd
....
```

or like shared buffers as below:

```
....
7f8ce656a000-7f8cef300000 rw-s 00000000 00:04 7334558
/dev/zero (deleted)
Size: 144984 kB
Rss: 75068 kB
Pss: 38025 kB
Shared_Clean: 0 kB
Shared_Dirty: 73632 kB
Private_Clean: 0 kB
Private_Dirty: 1436 kB
```

```
Referenced: 75068 kB
Anonymous: 0 kB
AnonHugePages: 0 kB
Swap: 0 kB
KernelPageSize: 4 kB
MMUPageSize: 4 kB
Locked: 0 kB
VmFlags: rd wr sh mr mw me ms sd
....
```

Acronym	Full form	Description
PSS	proportional set size	• For each private allocated memory chunk we count its size as is • We divide the size of a shared memory chunk by the number of processes that use it • $\sum_{pid} PSS(pid) =$ total memory used • We can get the size of private memory of a process this way:
Private		we need to count only private memory used by a backend or a worker, as all the shared is allocated by postmaster on startup. `$ grep '^Private' /proc/7194/smaps|awk '{a+=$2}END{print a*1024}'` `7852032` Private from psql `do $do$ declare` `l_command text :=` `$p$ cat /proc/$p$ || pg_backend_pid() || $p$/smaps $p$ ||` `p	grep '^Private' p ||` `p	awk '{a+=$2}END{print a * 1024}' p;` `begin` `create temp table if not exists z (a int);` `execute 'copy z from program ' || quote_literal(l_command);` `raise notice '%', (select pg_size_pretty(sum(a)) from z);` `truncate z;` `end;` `do;`