RLS Performance and Best Practices #14576

Locked GaryAustin1 announced in Troubleshooting

GaryAustin1 on May 25, 2023 Maintainer edited by supabase-search bot

This is a copy of a troubleshooting article on Supabase's docs site. It may be missing some details from the original. View the original article.

RLS performance and best practices

Although most of the time spent on thinking about RLS is to get it to handle security needs, the impact of it on performance of your queries can be massive.
This is especially true on queries that look at every row in a table like for many select operations and updates.
Note that queries that use limit and offset will usually have to query all rows to determine order, not just the limit amount so they are impacted too.

See the last section for ways to measure the performance of your queries as you test RLS improvements.

Is RLS causing performance issues (on a single table query)?

For very slow queries, or if using the tools at end of article, run a query with RLS enabled on the table and then with it disabled (this should only be done in a non production environment). If the results are similar then your query itself is likely the performance issue. Although, remember any join tables in RLS will also need to run their RLS unless a security definer function is used to bypass them. You can also create a service_role client to run the query bypassing RLS if in a secure environment.

How to improve RLS performance.

The following tips are very broad and each may or may not help the specific case involved. Some changes, like adding indexes, should be backed out if they do not make a difference in RLS performance and you are not using them for filtering performance.

1. The first thing to try is put an index on columns used in the RLS that are not primary keys or unique already.

For RLS like:
`auth.uid() = user_id`
Add an index like:
`create index userid on test_table using btree (user_id) tablespace pg_default;`
Improvement seen over 100x on large tables.

2. Another method to improve performance is to wrap your RLS queries and functions in select statements.

This method works well for JWT functions like `auth.uid()` and `auth.jwt()` as well as any other functions including security definer type.
Wrapping the function in some SQL causes an `initPlan` to be run by the optimizer which allows it to "cache" the results versus calling the function
on each row.
WARNING: You can only do this if the results of the query or function do not change based on the row data.
For RLS like this:
`is_admin() or auth.uid() = user_id`
Use this instead:
`(select is_admin()) OR (select auth.uid()) = user_id`

3. Do not rely on RLS for filtering but only for security.

Instead of doing this (JS client example):
`.from('table').select()`
With an RLS policy of:
`auth.uid() = user_id`
Add a filter in addition to the RLS:
`.from('table').select().eq('user_id',userId)`

4. Use security definer functions to do queries on other tables to bypass their RLS when possible.

Category
Troubleshooting

Labels
None yet

1 participant

Events
GaryAustin1 Locked 2 years ago

Instead of having this RLS where the roles_table has an RLS select policy of `auth.uid() = user_id`:
`exists (select 1 from roles_table where auth.uid() = user_id and role = 'good_role')`
Create a security definer function has_role() and do:
`(select has_role())` with code of `exists (select 1 from roles_table where auth.uid() = user_id and role = 'good_role')`
Note that you should wrap your security definer function in select if it is a fixed value per 2.
Remember functions you use in RLS can be called from the API.
Secure your functions in an alternate schema if their results would be a security leak.
Warning: If your security definer function uses row information as an input parameter be sure to test performance as you can't wrap the function as in 2.

5. Always optimize join queries to compare row columns to fixed join data.

Instead of querying on a row column in a join table WHERE, organize your query to get all
the column values that meet your query into an array or set.
Then use an IN or ANY operation to filter against the row column.
This RLS to allow select only for rows where the team_id is one the user has access to:
`auth.uid() in (select user_id from team_user where team_user.team_id = table.team_id)`
will be much slower than:
`team_id in (select team_id from team_user where user_id = auth.uid())`
Also consider moving the join query to a security definer function to avoid RLS on join table:
`team_id in (select user_teams())`
Note that if the `in` list gets to be over 10K items, then extra analysis is likely needed. See this follow up testing:
https://github.com/GaryAustin1/RLS-Performance/tree/main/tests/Supabase-Docs-Test .

6. Use role in TO option or roles dropdown in the dashboard.

Never just use RLS involving `auth.uid()` or `auth.jwt()` as your way to rule out 'anon' role.
Always add 'authenticated' to the approved roles instead of nothing or public.
Although this does not improve the query performance for the signed in user it does
eliminate 'anon' users without taxing the database to process the rest of the RLS.

Sample results

The code used for the below tests can be found here: LINK:
The tests are doing selects on a 100K row table. Some have an additional join table.

Show RLS and before after for above examples.

Test	RLS Before	RLS After	SQL	SQL
1	auth.uid()=user_id	user_id indexed	171ms	<.1
2a	auth.uid()=user_id	(select auth.uid()) = user_id	179	9
2b	is*admin() _table join*	(select is*admin()) _table join*	11,000	7
2c	is_admin() OR auth.uid()=user_id	(select is_admin()) OR (select auth.uid()=user_id)	11,000	10
2d	has_role()=role	(select has_role())=role	178,000	12
2e	team_id=any(user_teams())	team_id=any(array(select user_teams()))	173000	16
3	auth.uid()=user_id	add .eq or where on user_id	171	9
5	auth.uid() in *table join on col*	col in *table join on auth.uid()*	9,000	20
6	No TO policy	TO authenticated (anon accessing)	170	<.1

Tools to measure performance

Postgres has tools to analyze the performance of queries. https://www.postgresql.org/docs/current/sql-explain.html
The use of explain in detail for query analysis is beyond the scope of this discussion.
Here we will use it mainly to get a performance metric to compare times.
In order to do RLS testing you need to setup the user JWT claims and change the running user to `anon` or `authenticated`.

```
set session role authenticated;
set request.jwt.claims to '{"role":"authenticated", "sub":"5950b438-b07c-4012-8190-6ce79e4bd8e5"}';

explain analyze SELECT count(*) FROM rlstest;
set session role postgres;
```

This will return results like:

```
Seq Scan on rlstest  (cost=0.00..4334.00 rows=1 width=35) (actual time=170.999..170.999 rows=0 loops=1)
"  Filter: ((COALESCE(NULLIF(current_setting('request.jwt.claim.sub'::text, true), ''::text),
((NULLIF(current_setting('request.jwt.claims'::text, true), ''::text))::jsonb ->> 'sub'::text)))::uuid =
user_id)"
  Rows Removed by Filter: 100000
```

```
Planning Time: 0.216 ms
Execution Time: 171.033 ms
```

In this case the execution time is the critical number we need to compare.

PostgREST allows use of explain to get performance information on your queries with Supabase clients.

Before using this feature you need to run the following command in the Dashboard SQL editor (should not be used in production):

```
alter role authenticator set pgrst.db_plan_enabled to true;
NOTIFY pgrst, 'reload config';
```

Then you can use the .explain() modifier to get performance metrics.

```
const { data, error } = await supabase
  .from('projects')
  .select('*')
  .eq('id', 1)
  .explain({ analyze: true })

console.log(data)
```

This will return a result like:

```
Aggregate  (cost=8.18..8.20 rows=1 width=112) (actual time=0.017..0.018 rows=1 loops=1)
  ->  Index Scan using projects_pkey on projects  (cost=0.15..8.17 rows=1 width=40) (actual time=0.012..0.0
rows=0 loops=1)
        Index Cond: (id = 1)
        Filter: false
        Rows Removed by Filter: 1
Planning Time: 0.092 ms
Execution Time: 0.046 ms
```

*These two GitHub discussions cover the history that lead to this analysis...

Stable functions do not seem to be honored in RLS in basic form
current_setting can lead to bad performance when used on RLS

Thanks Steve Chavez and Wolfgang Walther in those threads.

Added example of security definer function having select of a team table, comparing against a column in main table

The example is in the test data as test2f.
In this case we start with a 1M row table with a team_id column.
We have a 1000 row team table that has user_ids and team(s) they belong too.
Basic RLS for a select would be `team_id = ANY(user_teams())`
This case times out with over 3 minutes as 1M rows must be searched and the function is run each time on 1000 rows.
Changing to wrap the function (method 2) `team_id = ANY(ARRAY(select user_teams()))` is a big improvement but can still take seconds.
Adding an index to team_id is the big win, but only with the second case. Without, the index case still times out.

Some results:

Policy	Index	Main Rs	Team Rs	on 10 teams	100	500	note
=ANY(user_teams())	no	1M	1000	>2Min	>2Min	>2Min	TO or killed
=ANY(user_teams())	yes	1M	1000	>2Min	>2Min	>2Min	TO or killed
=ANY(ARRAY(select user_teams()))	no	1M	1000	170ms	700	3300	
=ANY(ARRAY(select user_teams()))	yes	1M	1000	2ms	3	3	
in(1,2,3...100)	no	1M	NA	130ms	142	x	baseline check
=ANY(ARRAY(select user_teams()))	yes	1M	10K	x	x	x	24ms (on 1K teams)

↑ 3

0 comments