Research: Which is the best caching strategy with GraphQL for a big relational database?

In this research project I will look to the 3 main places where you can cache data: Server Side, Client Side & CDN. Often you hear people say that you can’t cache with GraphQL or that it breaks caching as we know it with REST API’s. Although there are multiple ways to cache with GraphQL, so I decided to take a look at these ways and will give you my conclusions.

Disclaimer: I’ve tested these caching solutions on a small scale and will give you my subjective thoughts. The timing results are taken in my local dev environment. (MacBook PRO (M1 PRO)).

# Technologies & Frameworks I’ve used

Client Side

* Next.js
* Apollo Client v3
* Urql Client

Server Side

* TypeGraphQL
* Apollo Server
* Redis

CDN

* GraphCDN

Database

* MSSQL
* 10 GB database dump Stack Overflow (https://www.brentozar.com/archive/2015/10/how-to-download-the-stack-overflow-database-via-bittorrent/)

Tools

* Docker
* Node.js

# Success criteria

* GraphQL API for the Stack Overflow relational database.
* Backend service in a JavaScript environment to test on the server side.
* Frontend app in Next.js (React) to test the client side.
* Comparison of the 3 main caching strategies (Client, Server & CDN) with their pros and cons.
* Performance (speed) measurement result of big ‘complex’ queries in the different strategies.
* Manual to setup caching for Front & Backend in the technology I found the best to work with.

# Evaluation criteria

Speed

* How fast can data be fetched? \*
* What is the impact of the data size?
* What is the impact of the nesting / relational tree

Developer experience

* How easy is it to implement?
* Pros & Cons
* Costs
* Bundle sizes (Client-Side bundles)

Cache busting

* How can I control the data freshness?
* What happens with mutations?

\*speed: I measured this in the react component by setting a useState hook of the with current time as start time and calculated the time between start and the moment the data was fetched.

# Caching possibilities

These are the caching possibilities I’ve looked at. A big library that is missing on the client-side is Relay, due to the learning curve and time I’ve decided to focus more on other clients.

|  |  |  |
| --- | --- | --- |
| **Server side** | **Client side** | **CDN** |
| HTTP caching (with persisted queries) | Apollo Client v3 | GraphCDN |
| Redis response caching | Urql |  |

## Server side

To build a backend server I’ve used Express (Node.js web framework) in combination with TypeGraphQL. Express is easy to start with and lets you use your own structure. On the other hand, TypeGraphQL is a well know framework that makes using GraphQL with TypeScript straight forward.

### Redis

#### Implementation

Redis is an in-memory database where you can store Key Value pairs and other data types but, in my case, I’ve just used Key Value storage. Due to the fact Redis is an in-memory database I thought it was the perfect tool to cache responses.

At first, I wanted to cache the full response of my Queries but then I would work against the principles of GraphQL. Because every Query can ask different fields, it would not be a good idea to cache the full response from a Query because this is client specific.

My solution to this was to cache only the database response I’ve got returned by TypeORM. This way I returned the same data with the Redis cache as my database would return.

Text

Description automatically generatedText

Description automatically generated

In the above example you can see how I first try to receive data from the cache, if not, I will fetch the data from the database.

This way was very effective and made it also possible to cache field resolvers (non-scalar types).

Text

Description automatically generated

#### Evaluation

##### Speed

#### A picture containing timeline Description automatically generated

As you can see in the above table, Redis is almost 50% faster than using no cache when you are fetching data from the frontend. This percentage is calculated on the average of 4 different queries, that represent a real-world example.

Redis has het most impact when you have large data and deep nested data, because these are expensive jobs for a database. Smaller queries can be easily handled by SQL servers so the impact there is much smaller.

##### Developer experience

Redis was a real pleasure to implement for response caches, although that’s not the only part that comes to play with caching. Like how do we handle cache busting?

Because Redis does not know about our relations when it stores our json’s as a string it’s much harder to control updates on the data. You can for example delete keys that contain a certain id. But what if the id of our updated or deleted entity does is not represented in the key? We would need to store every single entity to invalidate it when there are changes. This is a bit more complex to set up and slows down the backend while computing al these things. In client-side libraries like Apollo this can be done for you, without you needing to make a mechanism yourself.

I would recommend using Redis for large datasets that don’t change often. It will be effective, and you don’t need to worry about invalidation of your cache.

**Positive**

* Easy to implement for responses
* Usable in microservices architecture
* Fast

**Negative**

* Complex when you want to cache normalized data and keep track of their validation.

### HTTP Caching

GraphQL breaks HTTP caching because it only uses POST requests, right? Then you haven’t heard about **Automatic persisted queries!**

Automatic persisted queries are basically queries that are sent as a hash. If your server supports this, you can send your query as a GET request with a hash of your query in the query string of the request. How this exactly works can you find here: <https://www.apollographql.com/docs/apollo-server/performance/apq/> .

#### Implementation

If you don’t exactly know how HTTP caching works, I would recommend watching this video: <https://www.youtube.com/watch?v=HiBDZgTNpXY> to fully understand what is going on. By using Automatic persisted queries with GET requests, you can set cache-control headers in your responses. This enables you to let your browser or a CDN cache your responses. This can be done in 2 certain ways when you are using TypeGraphQL.

##### 1 Apollo Directives

The first one is by using the cacheControl directive from Apollo Server. With this directive you can define a max-age and scope to a type and a resolver. How this directives are used to calculate the cache-control header can your read over here: <https://www.apollographql.com/docs/apollo-server/performance/caching/#setting-cache-hints>

Because Apollo Sever only lets you set the max-age and scope (public or private) directives you can’t take fully advantage of HTTP caching. You cannot set for example the “stale-while-revalidate” directive, which is important for cache invalidation. To do so, you need to use way 2.

If you want to use TypeGraphQL like I did, you will need to define the directive @Directive (“”). You can see how in the example below.

Examples:

In the type:

Text

Description automatically generated

In a resolver:

Text

Description automatically generated

##### 2 Set the response header ‘cache-control’ manually.