Onderbeke Niels

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

Table of Contents

[Intro 2](#_Toc93851407)

[Technologies & Frameworks I’ve used 2](#_Toc93851408)

[Success criteria 3](#_Toc93851409)

[Evaluation criteria 3](#_Toc93851410)

[Caching strategies 4](#_Toc93851411)

[Server side 5](#_Toc93851412)

[Response caching with Redis 5](#_Toc93851413)

[Implementation 5](#_Toc93851414)

[Evaluation 6](#_Toc93851415)

[Speed 6](#_Toc93851416)

[Developer experience 7](#_Toc93851417)

[HTTP Caching 8](#_Toc93851418)

[Implementation 8](#_Toc93851419)

[Way 1: Apollo Directives 8](#_Toc93851420)

[Way 2: Set the response header ‘cache-control’ manually. 9](#_Toc93851421)

[Evaluation 11](#_Toc93851422)

[Speed 11](#_Toc93851423)

[Developer experience 11](#_Toc93851424)

[Client-side 12](#_Toc93851425)

[Apollo Client v3 – In Memory Cache 12](#_Toc93851426)

[Implementation 12](#_Toc93851427)

[Evaluation 13](#_Toc93851428)

[Speed 13](#_Toc93851429)

[Developer experience 14](#_Toc93851430)

[What about the bundle size? 15](#_Toc93851431)

[CDN (Content Delivery Network) 17](#_Toc93851432)

[GraphCDN 17](#_Toc93851433)

[Implementation 17](#_Toc93851434)

[Evaluation 18](#_Toc93851435)

[Speed 18](#_Toc93851436)

[Developer experience 18](#_Toc93851437)

[Results and conclusions 20](#_Toc93851438)

[Measurements 20](#_Toc93851439)

[Recommendations 21](#_Toc93851440)

[Overall 21](#_Toc93851441)

[Frontend 21](#_Toc93851442)

[Backend 22](#_Toc93851443)

[CDN 22](#_Toc93851444)

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

# Intro

As a last year Web Development student at Howest, I had the task to do research on a topic related to my studies. We had to formulate a research question and, in this article, I will try to give you an answer to that question.

In this research project I’ve looked to the 3 main domains where you can cache data. These being server-side, client-side & at a CDN.

Often 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 data with GraphQL. So, I decided to look at these methods, and I will give you my own subjective conclusions.

Disclaimer: I’ve tested these caching methods 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)) using Next.js.

## Technologies & Frameworks I’ve used

Client-side

* Next.js
* Apollo Client v3
* Urql Client

Server-side

* TypeGraphQL
* TypeORM
* 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 the cached data be fetched? \*
* How big is the impact of the data size?
* How big is the impact of the nesting / relational tree?

Developer experience

* How easy is it to implement / get started?
* Pros & Cons
* Costs
* Bundle sizes (client-side bundles)
* How can I control cache busting?

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

# Caching strategies

These are the caching strategies I’ve looked at. I know that there are many other libraries in other languages, but I’ve chosen to do my research in a JavaScript environment. A well-known library I did not test 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 automatic persisted queries) | Apollo Client v3 | GraphCDN |
| Response caching with Redis | 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-known framework that makes using GraphQL with TypeScript straight forward.

## Response caching with Redis

To handle in memory response caching on the server-side I’ve used Redis. Redis is super flexible and is easy to set up. An alternative could be Memcached for example.

### 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. The library I’ve used to connect with Redis is called **“ioredis”.**

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 present, I will fetch the data from the database. To set a max-age on the Key Value pairs you can just pass a time-to-live in seconds.

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

Text

Description automatically generated

### Evaluation

#### Speed

Table

Description automatically generated with low confidence

As you can see in the above table, Redis is almost 50% faster than using no caching 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. But because I’ve stored the full response as a stringified JSON I didn’t have control over the relations.

#### Developer experience

Redis was a real pleasure to implement for response caching, 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 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.

**Pros**

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

**Cons**

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

## 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. You can find how this actually works on this link: <https://www.apollographql.com/docs/apollo-server/performance/apq/> .

A nice thing is that this is not only available with Apollo! Urql client also supports this, so you don’t have to worry about creating this mechanism yourself.

### 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 directives in your HTTP cache-control header.

Your server keeps those hashes in memory to know which hash represents which query. You have the possibility to use another memory cache like Redis than the default from Apollo to cache those hashes. This caching method enables you to let a browser or CDN cache your API responses. This can be done in 2 certain ways when you are using TypeGraphQL.

#### Way 1: Apollo Directives

The first way is by using the cacheControl directive from Apollo Server. With this directive you can define a max-age and scope to a type or a resolver. How those directives are used to calculate the cache-control header can you read on this page: <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 the “stale-while-revalidate” directive for example, which is important to handle cache busting. To do so, you need to use way 2.

If you want to place directives on types and fields in your classes with **TypeGraphQL** like I did, you will need to define the directive @Directive("@cacheControl(maxAge: n)")

You can see how to do this in the example below.

Examples:

In the type (also possible at a field):

Text

Description automatically generated

In a resolver:

Text

Description automatically generated

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

Because I’m using Express, I can easily pass the response argument to my resolvers using the GraphQL context. This way I was able to set my own cache-control directives in a resolver.

How I’ve set this up:

index.ts:

Text

Description automatically generated

Resolver:

Text

Description automatically generated

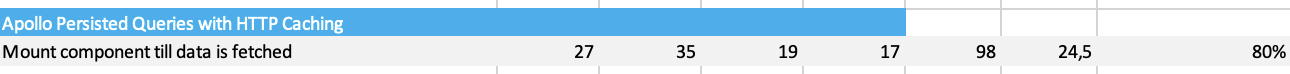
This way is very customizable & handy if you are familiar with HTTP caching!

### Evaluation

#### Speed

From my own testing, this way of caching seems to be the fastest. Because the cache data is stored locally, your browser has instant access to it. When your browser is using the local cache it’s on average 80% faster than working without caching! I did not measure this way of caching with a CDN, so this data is only based on the cache from your browser.

Table

Description automatically generated

#### Developer experience

Just like Redis I would recommend using Automated Persisted Queries for large responses that don’t change that often. It’s also handy if you already use a CDN provider for a REST API and you want to add a GraphQL API.

A good basic knowledge of HTTP caching is needed to use the full potential of this caching method. You also must understand how Apollo calculates the max-age and scopes when you are placing directives on types and fields. Like I said there are multiple ways to configure this, because of that, it can be a bit hard to get started with this.

**Pros**

* Easy to implement for responses
* Super-fast
* Usable with a CDN
* Server does not store anything

**Cons**

* Can be hard to set this up with your server
* Good knowledge about HTTP caching is needed

# Client-side

To test caching in a client-side environment I’ve chosen to use Next.js as framework, it’s easy to use and has many libraries that are made for it.

I will mainly talk about Apollo Client in this section. For my research I skipped Relay because it has a longer learning curve, but I did look at Urql. Because these Clients are very similar, I will just mention the main differences & won’t make a separate section for Urql. I will compare some parts of Apollo with Urql & share my thoughts on it with you.

You can find a good feature comparison guide on this site from Urql: <https://formidable.com/open-source/urql/docs/comparison/>

## Apollo Client v3 – In Memory Cache

Important to know is that Apollo’s client is specifically made for a React environment and it uses hooks to fetch data. If you are familiar with React you will like this library for sure! On the other hand, Urql has implementations for other frameworks as well, like Vue & Svelte. So, if you don’t want to stick to React, Urql is a great option!

Apollo Client delivers out of the box In Memory caching. It’s easy to setup and configure. It uses a normalized cache to work with relational data and that way it can easily modify or merge the cache after a mutation. By letting you set different cache/fetch policies you can describe how every Query or Mutation must behave when it fetches data. How this exactly works can you read on these pages:

<https://www.apollographql.com/docs/react/caching/overview/>

<https://www.apollographql.com/docs/react/data/queries/#setting-a-fetch-policy>

### Implementation

To get started you just need install the npm package @apollo/client. Unlike Urql you don’t need to install “Exchanges” to use the normalized cache, which makes it easier to get started. How to use this In Memory cache is good documented on the site of Apollo. So, I recommend reading the docs before using it!

### Evaluation

#### Speed

Apollo client is the second fastest caching method I’ve tried. Of the measurements I did, it was on average 69% faster than when using no caching at all. It’s even 10% faster than using Urql, while both were using In Memory normalized caches in the browser.

Table

Description automatically generatedTimeline

Description automatically generated with low confidence

#### Developer experience

In my opinion this is the best solution for GraphQL caching, client-side and even in general. It has great Chrome Dev tools that make debugging your cache super clear.

Due to its customizable in memory normalized cache you can keep your back- and frontend in sync. After a mutation or query it automatically updates the cache when it receives the right information back (id & updated fields). Then merges the enitie(s) in the cache it already has.

The cache updates only happen automatically when its receiving fields of an entity it did not already have or while updating mutated fields. You must update the cache yourself when deleting or creating entities or when you can’t return the updated fields from an update mutation.

How do we update the cache manually then? You can configure what should happen with the cache in a callback. Read more about it here: <https://www.apollographql.com/docs/react/caching/cache-interaction/>

It’s also possible to configure **polling** or **refetching** for your data in the cache, these are 2 strategies Apollo Client provides you with. With these options you can also keep your cached data in sync with your server! <https://www.apollographql.com/docs/react/data/queries/#updating-cached-query-results>

Pros

* Fast
* Easy to setup
* Hooks for fetching data
* Great dev tools

Cons

* Less flexible as Urql in terms of bringing your own implementation as an Exchange
* Big bundle size

#### What about the bundle size?

When we take a look at [Bundlephobia](https://bundlephobia.com/), you will notice that @apollo/client is a bigger package in comparison with urql, the other GraphQL client I’ve tested.

|  |  |
| --- | --- |
| **@apollo/client** | **urql** |
|  |  |

This is a huge difference at first sight. But when I analyzed my 2 apps their bundles with the same functionality but both different clients, the results were way closer to each other than you would think. The actual size of the bundles doesn’t differ that much.

My take on this is because with Urql, you will need an extra package to make normalized caching work (@urql/exchange-graphcache). Before I analyzed my bundles, I wanted the caches to work the exact the same way so that I could make a valid comparison. Urql caches by default the data as documents and not as normalized entities like Apollo. This means Urql can’t update the cache by default without sending a new request to the server.

|  |  |
| --- | --- |
| **@apollo/client** | **urql** |
| Build output | |
|  |  |
| Bundle Analyzer ([@next/bundle-analyzer](https://www.npmjs.com/package/@next/bundle-analyzer)) | |
|  |  |
|  |  |
| Chrome network tab | |
|  |  |

If you are new to the GraphQL world I would highly recommend using Apollo client because you will have to think less about how things work together. On the other hand, Urql is great option too and gives you more flexibility. But as you can see in the measurements, Urql is still a bit slower than Apollo.

In the end it’s your choice and I think you will have a great experience with both. You will have to decide which suits the best for your own project.

# CDN (Content Delivery Network)

## GraphCDN

As you already hear in the name, GraphCDN is a CDN build for GraphQL API’s. It embraces the power of GraphQL and makes caching on a CDN level easy and naturally. Just like with automatic persisted queries, some knowledge about HTTP caching is needed. GraphCDN sets the cache-control headers like “s-max-age” & “stale-while-revalidate” for you, you just need to declare them in an easy-to-understand form!

Graphical user interface, text, application, email

Description automatically generated

<https://graphcdn.io/>

GraphCDN offers you wide customization for your own caching strategy! You can create caching rules for queries, types & fields. You can define what fields are needed for purging your cache and you can define cache scopes based on headers like “Authorization”.

Analytics, Performance monitoring and error logging are included when you are using GraphCDN, you can track all the traffic that happens to your API. There are nice graphs and summarizations available to track how efficient your cache configuration is!

### Implementation

Setting up GraphCDN for your GraphQL API is easy, they have a great onboarding experience! You just need to fill in the URL of your GraphQL API and that’s it connection wise. After that, you need to define the caching rules you want.

You will get a URL from GraphQL that will handle your API traffic. If you want to, you can set this URL to your own domain name.

Before using GraphCDN I would recommend watching this video: <https://www.youtube.com/watch?v=EjrJtp4JaGQ&t=1061s> . In this video the co-founder of GraphCDN (Max Stoiber) talks about how it works & why it exists!

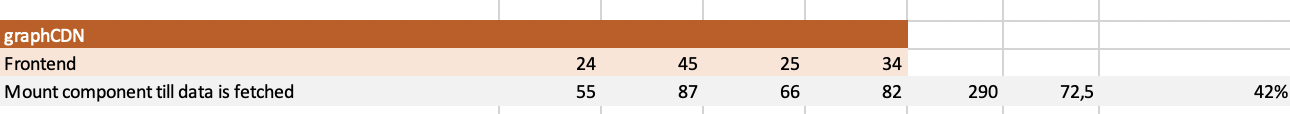
### Evaluation

#### Speed

From a speed standpoint GraphCDN is the slowest caching method that I’ve tested in my research. But it was still 42% faster than using no caching at all. But speed is not everything when it comes to caching.

CDN’s are built to deliver data as close to the consumers as possible. CDN’s are also solving the problem of your server having to deal with many requests at one time. They try to make the load on your server as low as possible by providing cached data. So, if you have a huge user base that’s divided around the world, GraphCDN is the way to go! Something to take in mind is that I tested this in my local environment, so that’s not the environment GraphCDN will show its strengths.

Table

Description automatically generated

#### Developer experience

GraphCDN provides a CLI so you can proxy requests to your local develop environment for testing your backend with your GraphCDN configuration. This is sweet because this way you won’t waste your free requests per month.

Talking about number of requests, GraphCDN is free up to 5 million requests per month. So, if you want to use GraphCDN in your production environment, the best option is to define your needs & compare them with the plans GraphCDN provides.

To implement cache busting / invalidate cached data after mutations, GraphCDN provides auto updates to renew your cache when an update mutation happens (if the updated data is included)! This is just like how Apollo Client handles updates. <https://docs.graphcdn.io/docs/automatic-cache-invalidation-via-mutations> .

They also provide a Purging API <https://docs.graphcdn.io/docs/purging-api> . This is a secured API that automatically gets created for your schema. It enables you to send requests from your backend server to purge caches that contain a certain Type, Field or Operation name. This is super handy to keep your data fresh in the cache! I would consider this a huge plus because you can relatively easy keep control over your caching strategy! Important to mention, because this is a public API that clears your real cache, you can’t really test it with your development setup using the CLI.

Nice to know, GraphCDN also support Automatic Persisted Queries!

Pros

* Super easy to get started
* Effects many users
* Develop environment with CLI
* Can be combined with HTTP caching headers that are set on the server-side
* Auto invalidation
* Purging API
* Build-in analytics, performance logging & error logging

Cons

* Paying after 5 million requests
* Analytics site sometimes a bit buggy

# Results and conclusions

## Measurements

Chart, bar chart

Description automatically generated

Graphical user interface, calendar

Description automatically generated with medium confidence

## Recommendations

I’ve ranked the different caching methods from 1 to 5 on the criteria that are important to me. 5 being the “best” and 1 being the “worst”.

### Overall

Chart, radar chart

Description automatically generated

### Frontend

Chart, radar chart

Description automatically generated

### Backend

Chart, radar chart

Description automatically generated

### CDN

Chart, radar chart

Description automatically generated