Onderbeke Niels

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

Table of Contents

[Intro 2](#_Toc93674579)

[Technologies & Frameworks I’ve used 2](#_Toc93674580)

[Success criteria 3](#_Toc93674581)

[Evaluation criteria 3](#_Toc93674582)

[Caching strategies 4](#_Toc93674583)

[Server side 5](#_Toc93674584)

[Response caching with Redis 5](#_Toc93674585)

[Implementation 5](#_Toc93674586)

[Evaluation 6](#_Toc93674587)

[Speed 6](#_Toc93674588)

[Developer experience 7](#_Toc93674589)

[HTTP Caching 8](#_Toc93674590)

[Implementation 8](#_Toc93674591)

[Way 1: Apollo Directives 8](#_Toc93674592)

[Way 2: Set the response header ‘cache-control’ manually. 9](#_Toc93674593)

[Evaluation 11](#_Toc93674594)

[Speed 11](#_Toc93674595)

[Developer experience 11](#_Toc93674596)

[Client-side 12](#_Toc93674597)

[Apollo Client v3 – In Memory Cache 12](#_Toc93674598)

[Implementation 12](#_Toc93674599)

[Evaluation 12](#_Toc93674600)

[Speed 12](#_Toc93674601)

[Developer experience 13](#_Toc93674602)

[What about the bundle size? 14](#_Toc93674603)

[CDN (Content Delivery Network) 16](#_Toc93674604)

[GraphCDN 16](#_Toc93674605)

[Implementation 16](#_Toc93674606)

[Evaluation 17](#_Toc93674607)

[Speed 17](#_Toc93674608)

[Developer experience 17](#_Toc93674609)

[Final verdict 19](#_Toc93674610)

[Measurements 19](#_Toc93674611)

[Recommendations 20](#_Toc93674612)

[Overall 20](#_Toc93674613)

[Frontend 20](#_Toc93674614)

[Backend 21](#_Toc93674615)

[CDN 21](#_Toc93674616)

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

# Intro

In this research project I’ve looked 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 look at these methods, and I will give you my own subjective 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)). using Next.js.

## 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)
* How can I control cache busting?
* 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 strategies

These are the caching strategies 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.

## Response caching with 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. 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, 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

Table

Description automatically generated with low confidence

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/> .

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

### 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 headers. 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 to cache those hashes, so you don’t overuse the memory of your API server. This caching method enables you to make your browser or a CDN cache your data 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 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 place directives on Types and Fields in your classes with TypeGraphQL like I did, you will need to define the directive @Directive (“”). You can see how 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 cach-control directives in a resolver.

How I set this up:

index.ts:

Text

Description automatically generated

Resolver:

Text

Description automatically generated

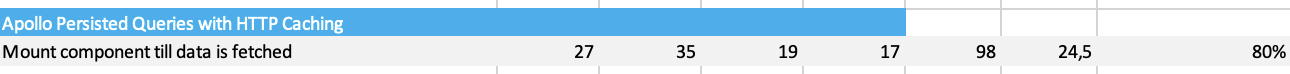
This way is very customizable & handy if you are familiar with HTTP caching!

### 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 cache! 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 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 Type 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.

**Positive**

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

**Negative**

* 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 frontend environment I’ve chosen to use Next.js as framework, it’s easy to use and many libraries are made for or compatible with react, so are caching libraries.

## Apollo Client v3 – In Memory Cache

This is one of the two client libraries I’ve tested for client-side caching. 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!

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 those 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 client 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 while implementing it!

### Evaluation

#### Speed

Apollo client is the second fastest caching method I’ve tried, from my measurements it was on average 69% faster than using no cache. It’s even 10% faster than using Urql client, when both are using In Memory 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. It has great Chrome Dev tools that makes checking and debugging your cache super clear. Due to its highly customizable normalized in memory 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) and merges it in the cache it already has.

The cache updates only happen automatically when receiving fields of an entity it did not already have or 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 do 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, this are 2 strategies Apollo provides you with. This way you can also keep your cached data in sync with your server! <https://www.apollographql.com/docs/react/data/queries/#updating-cached-query-results>

Positive

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

Negative

* Less flexible as Urql
* Big package bundle

#### What about the bundle size?

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

|  |  |
| --- | --- |
| **@apollo/client** | **@urql/core** |
|  | Graphical user interface, application  Description automatically generated |

This 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 off the fact that with Urql you 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 a normalized cache like Apollo. This means Urql can’t update by default the cache without sending a request to the browser.

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

I would recommend using Apollo if you are new in the GraphQL world because you must think less 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.

# CDN (Content Delivery Network)

## GraphCDN

As you probably already thought, 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 a wide variety of customization for your caching strategy! You can create caching rules for queries, types & fields. You can define on what fields are needed for purging your cache & 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 you 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 for connection. After that you need to define the caching rules you want. You get a URL from GraphQL that will handle your API traffic. If you want to, you can even set this URL to your own domain name.

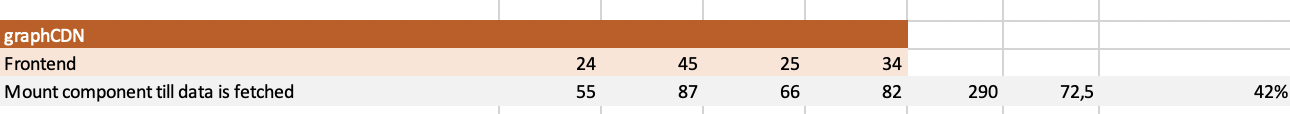
Before using GraphCDN I would recommend watching this video: <https://www.youtube.com/watch?v=EjrJtp4JaGQ&t=1061s> . In this video co-founder of GraphCDN (Max Stoiber) talks about how it works & why it exists!

### 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. 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 then you don’t waste your free requests per month. Talking about amount 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 options is to define your needs & compare with the plans GraphCDN provides.

To implement cache busting / invalidate cache after mutations, GraphCDN, has auto updates to update 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 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!

Positive

* 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

Negative

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

# Final verdict

## Measurements

Chart, bar chart

Description automatically generated

Graphical user interface, calendar

Description automatically generated with medium confidence

## Recommendations

### 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