What’s a resolver here?

What’s a resolver?

Well a function in a resolver needs to match the function in the graphql query itself

How to handle errors in graphql?

Handling errors in GraphQL is different from traditional REST APIs because the GraphQL server always responds with a 200 OK HTTP status, even when an error occurs.

- Using GraphQLErrorHandler implements the kickstarter’s GraphQlErrorHandler here

So this will be used instead of DefaultGraphQLExceptionHandler (the native one)

- And that’s one of them here, and then we have the next fetching data here,

And then you have a direct error here, and then a debug

How do we solve the n + 1 problem?

GraphQL is a query language and runtime to build APIs that reduces data results to only what the user requests. GraphQL uses schemas to define data inputs and responses from a single endpoint to a GraphQL runtime. The schemas allow clients to request specific information, so the responses will only include what the client needs.

In GraphQL the client specifies the data to return, but not how to fetch it from storage. Sometimes, a query could lead to unintentional, excessive backend requests. The n+1 problem is a typical example of when this can happen in GraphQL.

The n+1 problem is when multiple types of data are requested in one query, but where n requests are required instead of just one. This is typically encountered when data is nested, such as if you were requesting musicians and the names of their album titles. A list of musicians can be acquired in a single query, but to get their album titles requires at least one query per musician: one query to get n musicians, and n queries to get a list of albums for each musician. When n becomes sufficiently large, performance issues and failures can arise. This is a common situation when using GraphQL because the client has full flexibility in building the queries.

In this article, we'll take a closer look at the n+1 problem and what it looks like in practice. We'll also get an overview of efficient techniques to avoid the problem and improve performance.

GraphQL server setup

GraphQL queries are made against a single endpoint. GraphQL servers like the Apollo Server allow the backend to define a GraphQL schema. The schema is a data model at the application layer that indicates how the client can query the database, just like a REST API contract.

When building a runtime in GraphQL, a unique resolver should be present for each discrete data type. The following example shows a sample Apollo Server setup, GraphQL schema and resolvers for different entity fields. The client can make calls against this runtime to get musician data, and also get album related data for the musician.

//Schema Definitions

const typeDefs = gql`

type Album {

id: ID!

title: String!

artistId: ID!

}

type Musician {

id: ID!

name: String!

albums: [Album]

}

type Query {

musicians: [Musician]

}

`;

//Resolver definitions

const resolvers = {

Query: {

musicians: () => {

//Database/API call to get a list of musicians

return musicians;

},

},

Musician: {

albums: (musician) => {

//Input is a single musician

//Database/API call to get a list of albums for this single musician

return albums

},

},

};

const server = new ApolloServer({ typeDefs, resolvers });

server.listen(3000).then(({ url }) => {

console.log(`Starting new Apollo Server at ${url}`);

});

Copy

The n+1 problem in GraphQL

The client creates requests against the GraphQL server, tailored to include the information needed for display. In the server we have set up above, we have allowed clients the flexibility to request a list of musicians, and inside the musician resolver we are allowing the client to fetch associated albums per musician.

query {

musicians {

id,

name,

albums {

title

}

}

}

What exactly is it?

The client query above requests some musician related data and albums related data for each musician. We know how the schema and resolvers of this GraphQL server are designed, and the way the server will execute this query can lead to issues. In this case, the server will first fetch the list of musicians.

Let’s say it finds n musicians in the database. For each musician found, the albums() resolver will be invoked to locate all the albums associated with that musician. This resolver will trigger a database call for each musician, which will be n calls. This means that in total, there will be n+1 database calls occurring. This is not very efficient and won’t scale after a point.

Consequences of the n+1 problem

The n+1 problem can lead to several client and server issues. The main problem is scaling, it might go unnoticed until a point, but as our application scales and n grows, the number of calls to the database or calls to the API that resolves our GraphQL fields will become unmanageable. Also, the latencies will start increasing, the product will behave inconsistently, pages without many nested calls will load quickly, while others that require nested data will be much slower. If you are using a cloud-based server like AWS or GCP to run the database, extra calls to the database will cost more in service fees and eventually all of it combined will lead to a very poor user experience. High latency reduces the ability to retain customers, and hence we should try to closely monitor it and fix the issues in our system as our application scales.

**How to solve n+1 in GraphQL**

Since this is a common issue with GraphQL, there are well-established solutions for handling it. These include using Data Loaders or Batching.

Data loaders in GraphQL

Data loaders are a way to solve the n+1 problem. They can batch similar client GraphQL requests into a single query. Basically, consider them as an intermediate layer that converts multiple similar requests to a single batched request. For our case, instead of making n different requests for getting albums of each musician, data loaders will make one request send an array of musician ids and receive an array of album objects.

Internal image\_ GraphQL n+1 problem.png

A screenshot of a computer

Description automatically generated

The implementation of data loaders depends on which version of GraphQL you are using. Some have built-in data loader functionality, like the java-dataloader for GraphQL Java.

GraphQL has a well maintained dataloader library that can be used in our GraphQL server. This utility mimics the original GraphQL calls with loaders passed to each resolver in the context value. The example below shows what the GraphQL query for musicians would look like with a data loader.

const DataLoader = require('dataloader');

// The dataloader takes in an array of musician ids and returns a promise that will return

// the album data for each musician.

const albumLoader = new DataLoader(musicianIds => {

// DB call that accepts list of musicianIds.

return databaseCall(musicianIds)

// OR it can be an API call that accepts a list of musicianIds.

return apiCall(musicianIds)

})

// Add the data loader to context

const context = async () => {

const loaders = {

album: albumLoader(musicianIds),

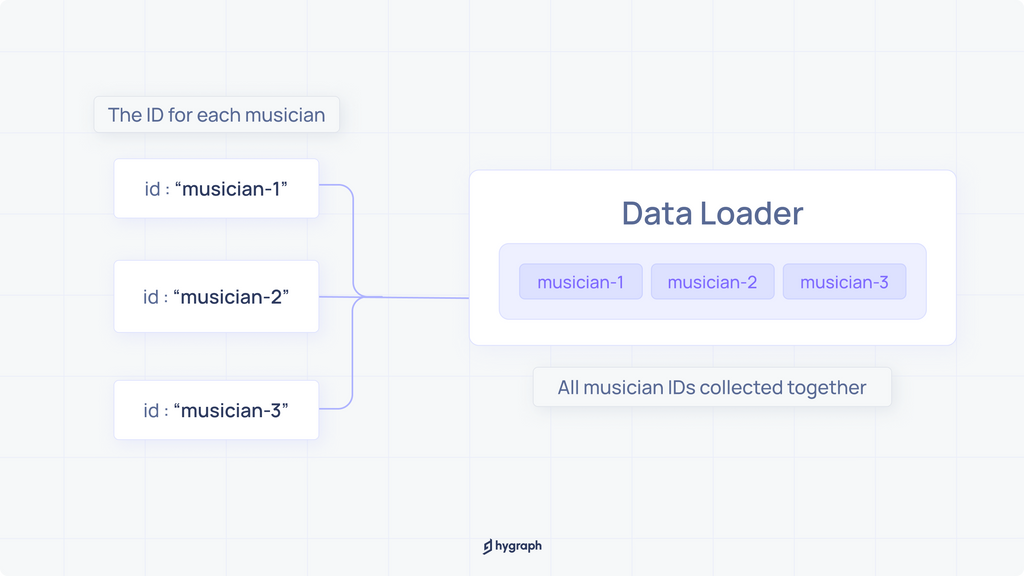
// Create more loaders for other data that is nested in your schema

}

return loaders;

}

// Use this context in the Apollo Server definition to pass to each resolver when executed.



Copy

Once the data loader is set up and available in the context, the resolver should be updated to use that loader. The loader is only triggered once, for the list of musicians fetched, so the number of DB / API calls will be reduced to two.

//Resolver definition

const resolvers = {

Query: {

// Destructure data loader from context.

musicians: (\_, args, { loaders }) => {

//Database/API call to get a list of musicians

return loaders.album(musicians.map(thisMusician => thisMusician.id));

}

}

};