

AUGUST 18, 2016

# GraphQL Concepts Visualized



Dhaivat Pandya  
Core Developer



@dhaivatsays

BASICS

GRAPHQL

LAST UPDATED MAY 19, 2021

GraphQL is often explained as a “unified interface to access data from different sources”. Although this explanation is accurate, it doesn’t reveal the underlying ideas or the motivation behind GraphQL, or even why it is called “GraphQL” — you can see the stars and the night, but not quite the “The Starry Night”.

The true heart of GraphQL lies in what I think of as the application data graph. In this article, I’ll introduce the app data graph, talk about how GraphQL queries operate on the app data graph and how we can cache GraphQL query results by exploiting their tree structure.

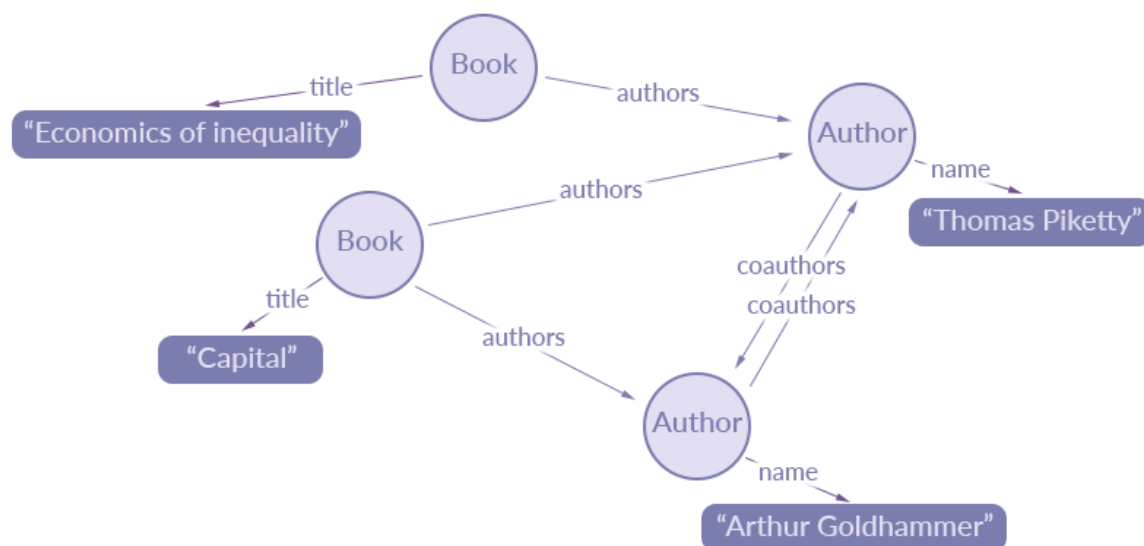
*Update 2/7/2017: You can now see this in talk form below!*

# The application data graph

A lot of data in modern applications can be represented using a graph of nodes and edges, with nodes representing objects and the edges representing relationships between these objects.

For instance, say we're building a cataloging system for libraries. To keep it simple, our catalog has a bunch of books and authors and each of these books has at least one author. The authors also have coauthors with whom the author has written at least one book.

If we visualize these relationships in the form of a graph, they look something like this:



The graph represents the relationships between the various pieces of data that we have and the entities (e.g. “Book” and “Author”) we’re trying to represent. Pretty much all applications operate on this kind of graph: they read from it and write to it. This is where GraphQL comes in.

### GraphQL allows us to extract trees from the app data graph.

That sounds pretty puzzling at first, but let’s break down what it means. Fundamentally, a tree is a graph which has a starting point (the root) and the property that you can’t trace your finger along the nodes through the edges and come back to the same node, i.e. it has no cycles.

## Traversing the Graph with GraphQL

Let’s take a look at an example of a GraphQL query and how it “extracts a tree” out of an application graph. Here’s a query we could run against the data

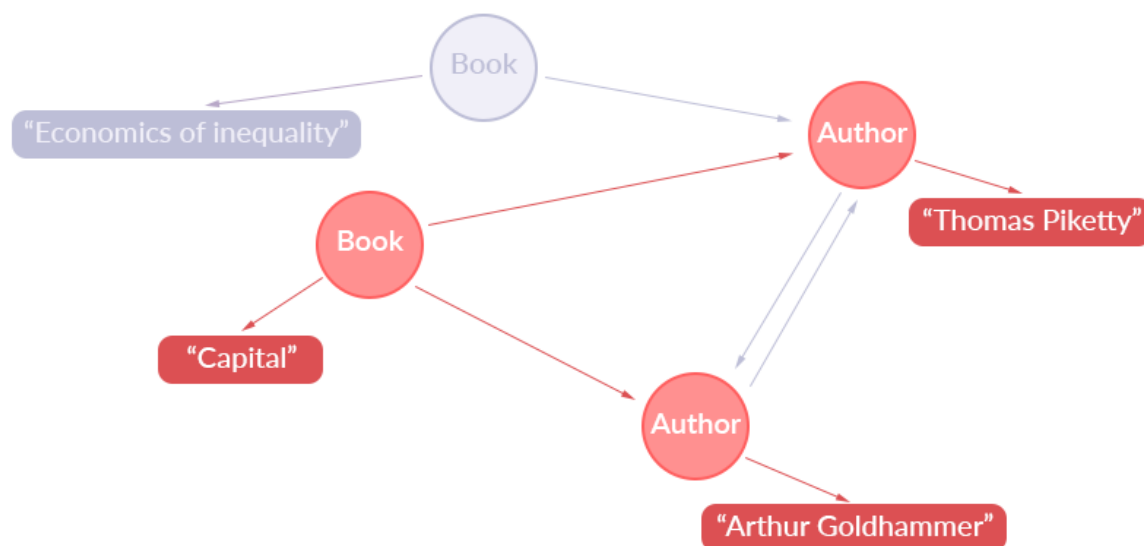
graph we were just talking about:

```
query {  
  book(isbn: "9780674430006") {  
    title  
    authors {  
      name  
    }  
  }  
}
```

Once the server resolves the query, it returns this query result:

```
{  
  book: {  
    title: "Capital in the Twenty First Century",  
    authors: [  
      { name: 'Thomas Piketty' },  
      { name: 'Arthur Goldhammer' },  
    ],  
  }  
}
```

Here's what that looks like in terms of the application data graph:

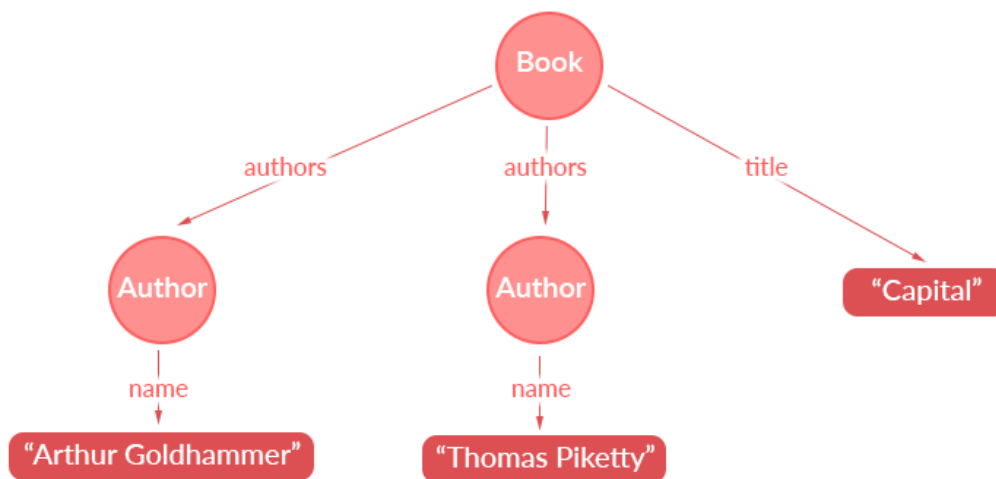


## Query Paths

Let's figure out how this information was actually extracted from the graph by the GraphQL query.

GraphQL allows us to define a *Root Query Type* (we'll refer to it as *RootQuery*), which defines where a GraphQL query can start when traversing the app data graph. In our example, we start with a "Book" node, which we've selected using its ISBN number with the query field `book(isbn: ...)`. Then, the GraphQL query traverses the graph by following the edges marked by each of the nested fields. For our query, it hops from the "Book" node to the node containing the string title of the book through the `title` field in the query. It also gets "Author" nodes by following the edges on the "Book" that are labelled with the `authors` field and gets each author's `name` as well.

To see how this result creates a tree, we just have to move around the nodes to make it look more like one:



For each piece of information that the query returns, there’s an associated query path, which consists of the fields in the GraphQL query that we followed to get to that information. For example, the book’s title “Capital” has the following *query path*:

“  
RootQuery → book(isbn: “9780674430006”) → title

The fields in our GraphQL query (i.e. *book*, *authors*, *name*) specify which edges should be followed in the application data graph to get our desired result. This is where GraphQL gets its name: **GraphQL is a query language that traverses your data graph to produce a query result tree.**

## Caching GraphQL results

To build a really snappy, fluid application that doesn’t spend most of its life

presenting a loading spinner to its users, we want to reduce roundtrips to the server by using a cache. **It turns out that the tree structure of GraphQL lends itself extremely well to client-side caching.**

As a simple example, suppose we have some code on your page that fetches the following GraphQL query:

```
query {  
  author(id: "8") {  
    name  
  }  
}
```

Later on, some other section of the page requests the same query again. Unless we absolutely need the newest data, this second query can be resolved with the data we already have! This means that the cache needs to be able to resolve queries without ever sending them to the server, making our application faster. But we can do much better than just caching the exact queries we have previously fetched.

Let's take a look at the approach Apollo Client takes to caching GraphQL results. Fundamentally, GraphQL query results are trees of information from your server-side data graph. We want to be able to cache these result trees to avoid reloading them every time we need them again. To do this, we make a key assumption:

Apollo Client assumes that each path within your application data graph, as specified by your GraphQL queries, points to a stable piece of information.

If this doesn't hold true in some cases (e.g. when the information pointed to by a particular query path changes really frequently), we can prevent Apollo Client from making this assumption with the concept of *object identifiers*, which we'll introduce later. But, in general, this turns out to be a reasonable assumption to make when it comes to caching.

## Same path, same object

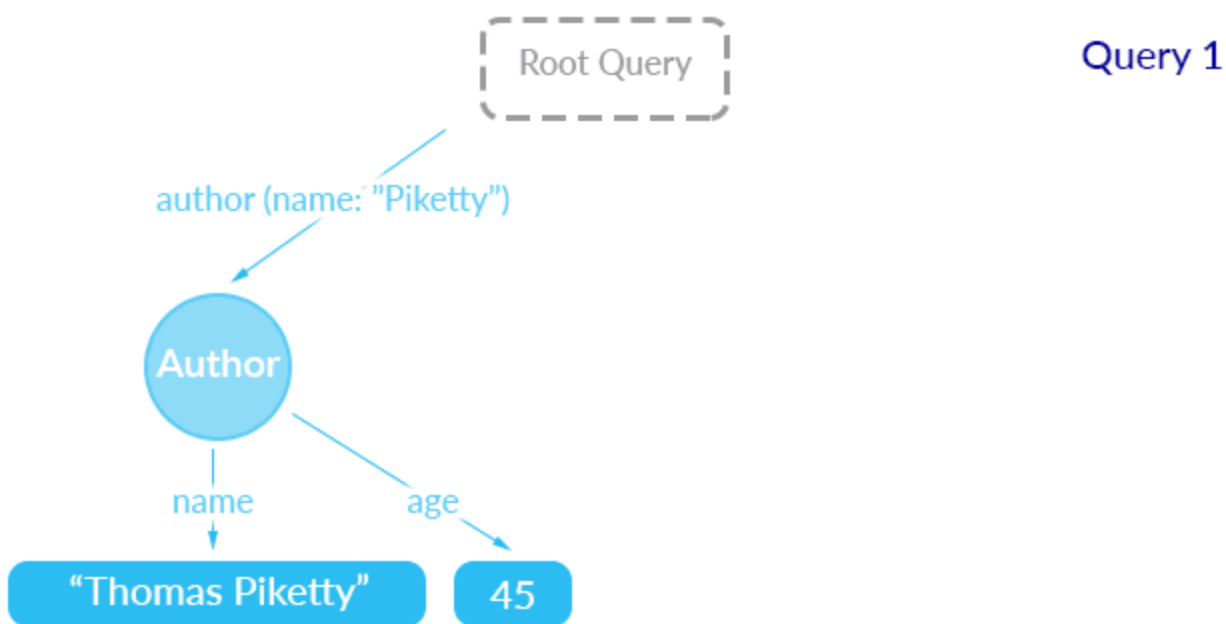
The assumption of “same path means same object” introduced by the last point is incredibly useful. For example, say we have these two queries, fired one after the other:

```
query particularAuthor {
  author(name: "Thomas Piketty") {
    name
    age
  }
}
query authorAndBook {
  book(isbn: "9780674430006") {
    title
  }
  author(name: "Thomas Piketty") {
    name
    age
  }
}
```

You can see just by looking at the queries that the second query doesn't need to go to the server to get the name of the author. This information can just be found within the cache from the result of the previous query.




Apollo Client uses this kind of logic to remove parts of the query based on the data already in the cache. It's able to do this kind of diffing because of the path assumption. It assumes that the path *RootQuery*→*author(id: 6)*→*name* would have fetched the same information in both queries. Of course, if this assumption doesn't work for you, you can override caching entirely by using the *forceFetch* option.



This assumption works really well because the query path also includes the arguments we use within GraphQL. For example...

“  
RootQuery → author(id: 3) → name

is different from



RootQuery → author(id: 6) → name

...so Apollo Client won't assume that they represent the same information and try to merge one with the result of the other.

## Use *object identifiers* when the path assumption isn't enough

It turns out that we can do even better than just following query paths from the root. Sometimes, you might access the same object through two totally different queries, giving that object two different query paths.

For example, given that each of our authors has some set of coauthors, we can end up accessing some "Author" objects through that field:

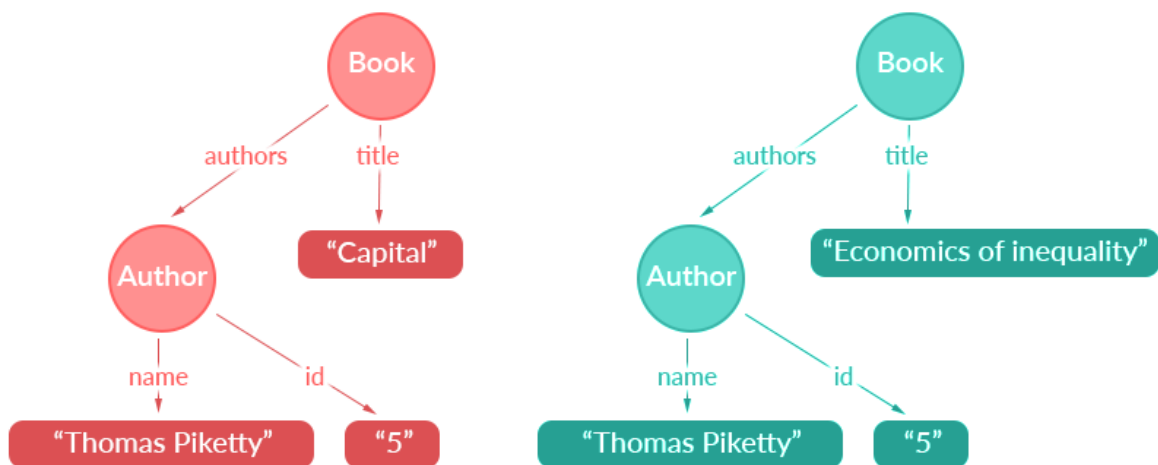
```
query {  
  author(name: "Arthur Goldhammer") {  
    coauthors {  
      name  
      id  
    }  
  }  
}
```

But we can also access an author directly from the root:

```
query {  
  author(id: "5") {  
    name  
    id  
  }  
}
```

Assume that the author with the name of “Arthur Goldhammer” and the author of ID 5 are coauthors on some book. Then, we’d end up saving the same information (i.e. the information about Thomas Piketty, the author with ID 5) twice in our cache.

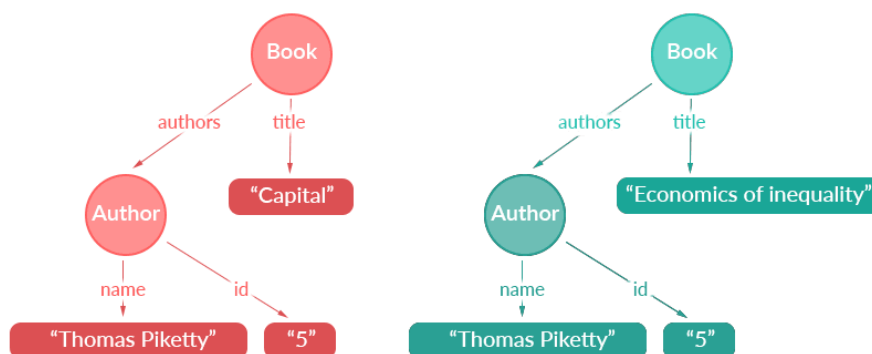
This is that would would like in our tree cache structure:



The problem is that both queries are referring to the same piece of information within the app data graph, but Apollo Client has no way to know that yet. To solve that issue, Apollo Client supports a second concept: object identifiers. Basically, you can specify a unique identifier for any object you

query. Then, **Apollo Client assumes that all objects with the same object identifier represent the same piece of information.**

Once Apollo Client knows that, it can rearrange the cache in a much nicer way:



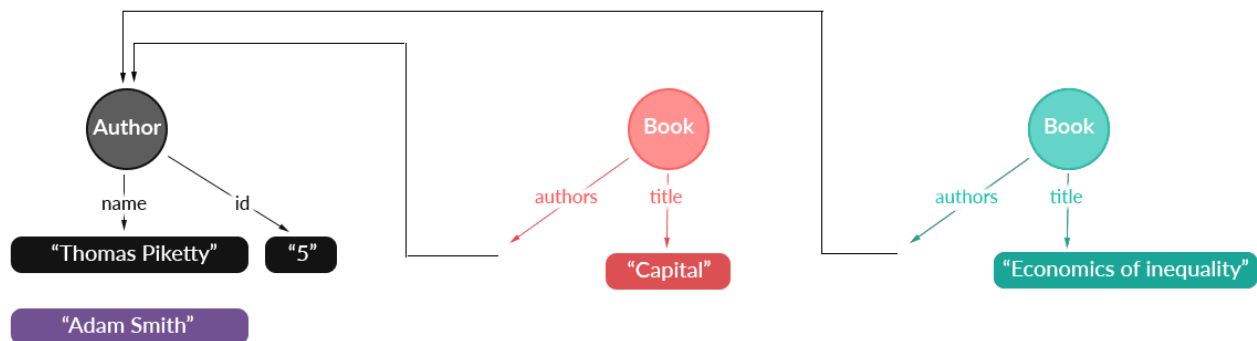
This means that object identifiers have to be unique across your whole app. As a consequence, you can't just use your SQL IDs directly, because then you might have an author with a SQL ID of 5 and a book with a SQL ID of 5. But this is easy to fix: to generate a unique object identifier, just append the `__typename` returned by GraphQL to the IDs generated by your backend. So, an author with a SQL ID of 5 could have an object identifier of `Author:5` or something similar.

## Keeping query results consistent

Continuing on with the last two queries we just dealt with, let's think about what happens if some data changes. For example, what if you fetch some other query and realize that the author with an ID of 5 has changed his/her name? What happens to the parts of your UI that currently refer to the old name that the author with an ID of 5 had?

Here's the awesome part: they will get updated automatically. That leads us to one more guarantee that Apollo Client provides: **If any of a watched query tree's nodes change in value, the query will be updated with the new result.**

So, in this case, we have two queries that both rely on the author with an object identifier of "Author:5". Since both query trees refer to this author, any update to the author's information will be propagated to both queries:



If you use a view integration like *react-apollo* or *angular2-apollo* with Apollo Client, you don't have to worry about setting this up: your components will just get the new data handed to them and re-render automatically. If you're not using a view integration, the core method *watchQuery* does the same thing by giving you an observable that is updated whenever the store

changes.

Sometimes it doesn't make sense for your application to have object identifiers for everything, or you might not want to deal with them directly in your code but still need particular bits of information in the cache to be updated. This is why we expose convenient but powerful APIs such as *updateQueries* or *fetchMore* that let you incorporate new information into these query trees with very granular control.

## Summary

The backbone of any application lies in the app data graph. Back in the day, when we had to roll our own HTTP requests to REST endpoints to get stuff into and out of this application graph, caching on the client was incredibly hard because data fetching was very application-specific. GraphQL, on the other hand, gives us lots of information we can use to automatically cache stuff.

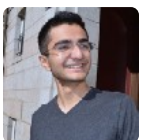
If you understand five simple concepts, you can understand how reactivity and caching (i.e. all the magic that makes your app fast and slick) work within Apollo Client. Here they are, restated:

1. GraphQL queries represent a way to get trees out of your app data graph. We call these *query result trees*.
2. Apollo Client caches *query result trees*. To do this, it applies two assumptions:

3. *Same path, same object* — the same query path usually leads to the same piece of information.
4. *Object identifiers when the path is not enough* — if two results are given the same object identifier, they represent the same node/piece of information.
5. If any cache node involved in a query result tree is updated, Apollo Client will update the query with a new result.

The above is all you need to know to be an expert in Apollo Client and GraphQL caching in general. Too much for one post? Don't worry — we'll keep posting more conceptual information like this when possible, so that everyone can understand the purpose behind GraphQL, where it gets its name, and how to reason clearly about any aspect of GraphQL result caching.

*Thanks to [Slava Kim](#) for the awesome diagrams and feedback on this post!*



WRITTEN BY

**Dhaivat Pandya**



Follow

[Read more by Dhaivat Pandya](#) →

## Stay in our orbit!

Become an Apollo insider and get first access to new features, best practices, and community events. Oh, and no junk mail. Ever.

[Subscribe](#)

## Make this article better!

Was this post helpful? Have suggestions? Consider [leaving feedback](#) so we can improve it for future readers ✨.

## SIMILAR POSTS

DECEMBER 13, 2022

### Add Python to your GraphQL API with GraphOS and Strawberry GraphQL



by Patrick Arminio



MARCH 3, 2022

## Using the GitHub GraphQL API with Apollo Studio Explorer



by Ceora Ford

JANUARY 11, 2022

## Getting Started with Vue Apollo



by Khalil Stemmler

### Company

About Us

Platform

Enterprise

Pricing

Customers

Careers

Team

### Community

Blog

Docs

GraphQL Summit

### Help

Get Support

Terms of Service

## Privacy Policy