CS188
Scalable Internet Services

John Rothfels, 11/12/20

Announcements

- Not doing AWS deployment this week. Starting next.
- You should have a demo-able app to deploy next Friday during lab.
- You can do the majority of your final project without deploying to AWS!
- Will provide detailed instructions for final paper tomorrow during lab.
    - Get ahead! Start writing your paper during labs.

For your projects...

- Use 1-3 different load tests.
- Start collecting results:
    - If you make a change to the app, do a load test before & after.
    - Save images, data, all relevant info you can think of (What computer was it running on? How much CPU/RAM? What feature was being tested? Etc.)
    - Keep yourself organized. Treat your project like a science experiment. Control variables so you are testing one thing at a time. Use the same load test script before and after. Run the code on the same machine, etc.
    - Analyze results each time. Make hypothesis to explain the data.
    - Do all comparisons against a common baseline!

Motivation

You’re building your project, and certain things aren’t working very well…

Investigation 🕵️‍♂️

Let’s use the k6 load testing tool to see which routes / graphql operations don’t work well under heavy load.

Investigation 🕵️‍♂️

Trying different k6 scenarios / executors may show us different results.

```
export const options = {
 scenarios: {
   example_scenario: {
     executor: 'constant-vus',
     vus: 1000,
     duration: '10s',
   },
 },
}
```

```
export const options = {
 scenarios: {
   example_scenario: {
     executor: 'ramping-vus',
     startVUs: 0,
     stages: [
       { duration: '60s', target: 5000 },
       { duration: '60s', target: 0 },
     ],
     gracefulRampDown: '0s',
   },
 },
}
```

Investigation 🕵️‍♂️

Let’s use a very simple user script to start:

```
export default function () {
 http.get('http://localhost:3000/')
 sleep(Math.random() * 3)
}
```

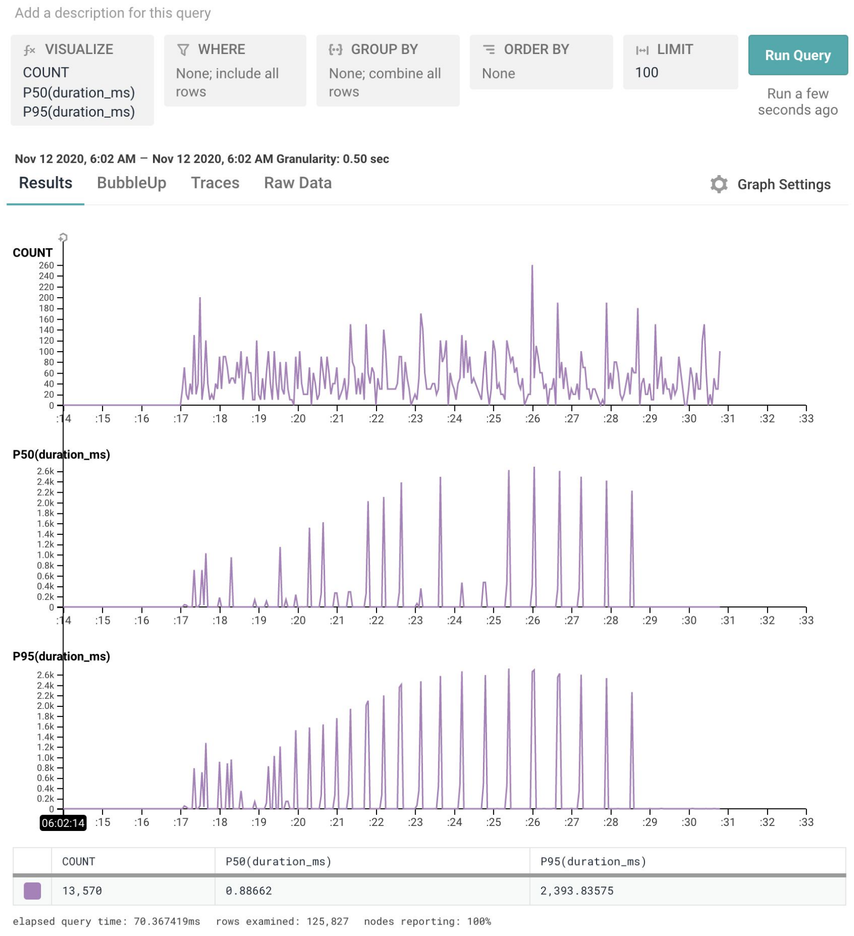
Investigation 🕵️‍♂️

And look at what happens in Honeycomb.

Investigation 🕵️

And look at what happens in honeycomb.

⁉️ **What are those crazy latency spikes?!**



Add a description for this query

VISUALIZE	WHERE	GROUP BY	ORDER BY	LIMIT

COUNT
P50(duration_ms)
P95(duration_ms)
	None; include all rows	None; combine all rows	None	100

Run Query

Run a few seconds ago

Nov 12 2020, 6:02 AM – Nov 12 2020, 6:02 AM Granularity: 0.50 sec

Results BubbleUp Traces Raw Data Graph Settings

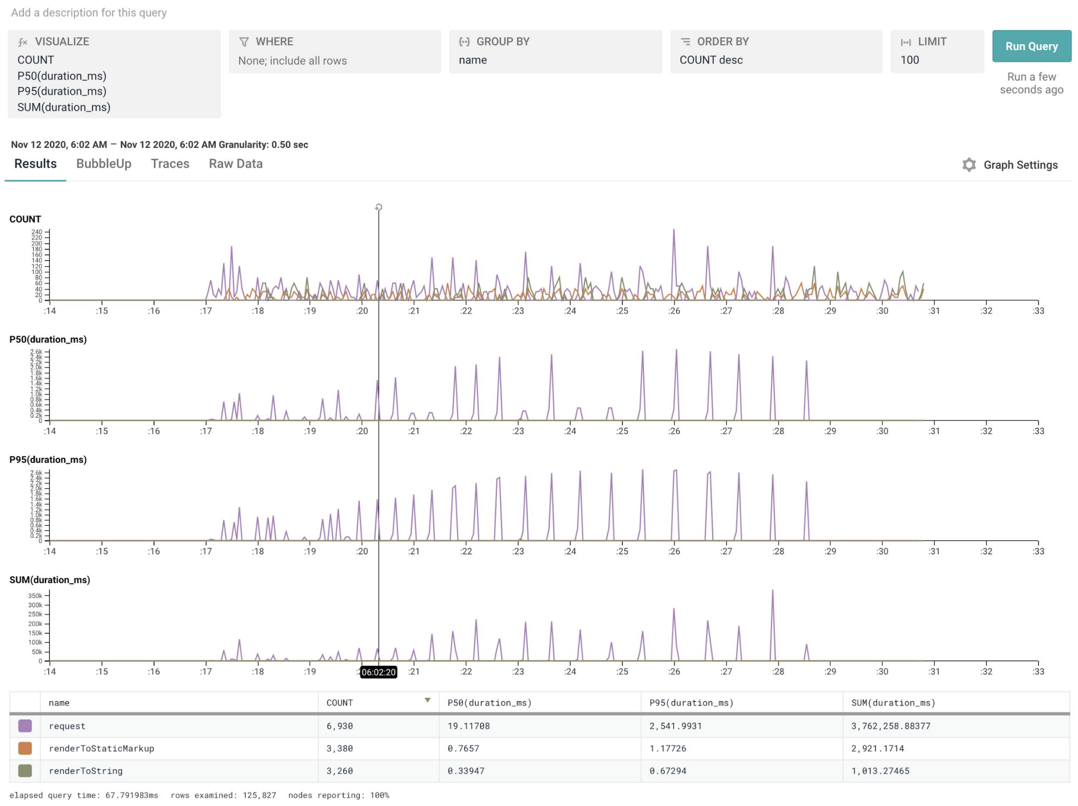
	COUNT	P50(duration_ms)	P95(duration_ms)
🟪	13,570	0.88662	2,393.83575

```
elapsed query time: 70.367419ms   rows examined: 125,827   nodes reporting: 100%
```

Investigation 🕵️

⁉️ Can we learn anything by grouping the data?

- The `request` event is taking the most time.



VISUALIZE: COUNT, P50(duration_ms), P95(duration_ms), SUM(duration_ms)
WHERE: None; include all rows
GROUP BY: name
ORDER BY: COUNT desc
LIMIT: 100

Nov 12 2020, 6:02 AM – Nov 12 2020, 6:02 AM Granularity: 0.50 sec

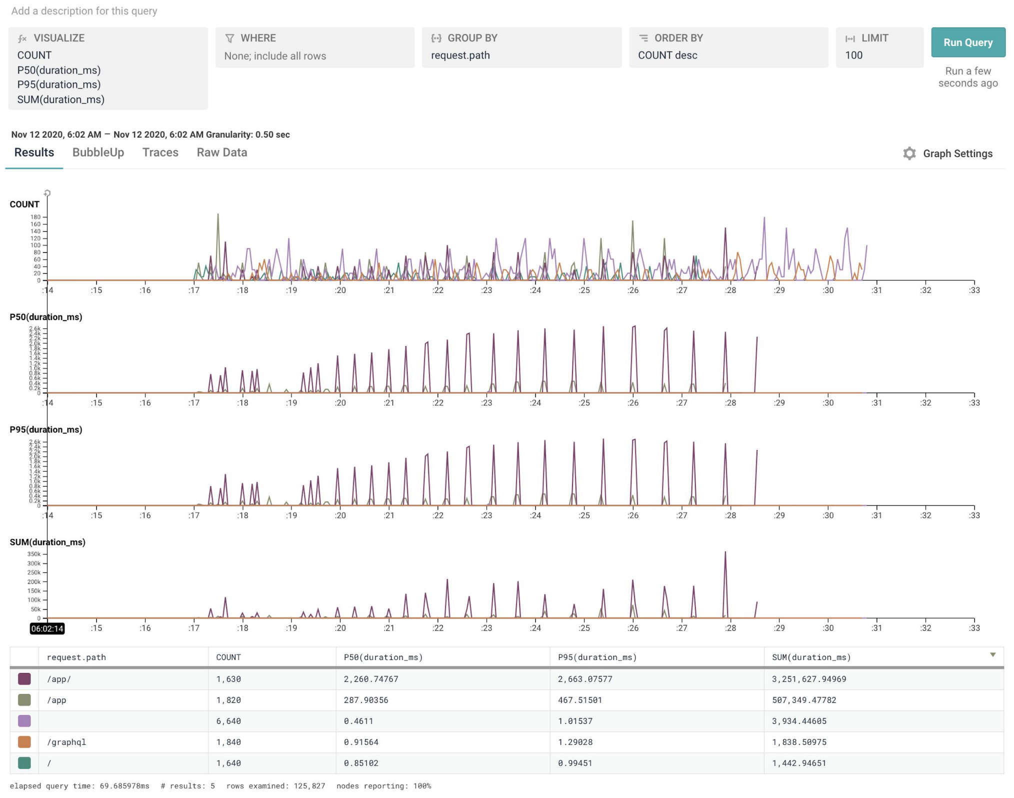
	name	COUNT	P50(duration_ms)	P95(duration_ms)	SUM(duration_ms)
🟪	request	6,930	19.11708	2,541.9931	3,762,258.88377
🟧	renderToStaticMarkup	3,380	0.7657	1.17726	2,921.1714
🟩	renderToString	3,260	0.33947	0.67294	1,013.27465

```
elapsed query time: 67.791983ms   rows examined: 125,827   nodes reporting: 100%
```

Investigation 🕵️

⁉️ Can we learn anything by grouping the data?

- We are hitting 4 different server routes with our load test!
- The `/app/` path is the one with most of the latency



Add a description for this query

VISUALIZE
COUNT
P50(duration_ms)
P95(duration_ms)
SUM(duration_ms)

WHERE
None; include all rows

GROUP BY
request.path

ORDER BY
COUNT desc

LIMIT
100

Run a few seconds ago

Nov 12 2020, 6:02 AM – Nov 12 2020, 6:02 AM Granularity: 0.50 sec

	request.path	COUNT	P50(duration_ms)	P95(duration_ms)	SUM(duration_ms)
🟪	/app/	1,630	2,260.74767	2,663.07577	3,251,627.94969
🟩	/app	1,820	287.90356	467.51501	507,349.47782
🟪		6,640	0.4611	1.01537	3,934.44605
🟧	/graphql	1,840	0.91564	1.29028	1,838.50975
🟩	/	1,640	0.85102	0.99451	1,442.94651

```
elapsed query time: 69.685978ms   # results: 5   rows examined: 125,827   nodes reporting: 100%
```

Investigation 🕵️‍♂️

⁉️ **Hypothesis? What’s going on to create the latency?**

Investigation 🕵️‍♂️

⁉️ **Hypothesis? What's going on to create the latency?**

- Connecting to (or querying) the DB?
- Redirect logic?
- Server-side rendering?

Investigation 🕵️

⁉️ **Hypothesis? What's going on to create the latency?**

- Connecting to (or querying) the DB?
    - This would only be happen if we sent a cookie / header indicating who the logged in user is

```
server.express.post(
 '/graphql', asyncRoute(async (req, res, next) => {
   const authToken = req.cookies.authToken || req.header('x-authtoken')
   if (authToken) {
     const session = await Session.findOne({ where: { authToken }, relations: ['user'] })
     if (session) {
       const reqAny = req as any
       reqAny.user = session.user
     }
   }
   next() // end of middleware, pass through to the next handler (api.ts)
 })
)
```

Investigation 🕵️‍♂️

⁉️ Hypothesis? What’s going on to create the latency?

- Connecting to (or querying) the DB?

To test, we can (1) delete the middleware that queries for users, or (2) authenticate k6 requests.

```
export default function () {
 http.get('http://localhost:3000/', { cookies: { authToken: '9f10c46a-98ff-4351-86bb-74a601eb5ac2' } })
 sleep(Math.random() * 3)
}
```

Investigation 🕵️

⁉️ **Hypothesis? What's going on to create the latency?**

- Redirect logic?
    - This only seems to be happening on the root (index) route

To test, we can modify the user script to avoid the redirect.

```
export default function () {
 http.get('http://localhost:3000/app/index', { cookies: { authToken: '9f10c46a-98ff-4351-86bb-74a601eb5ac2' } })
 sleep(Math.random() * 3)
}
```

Investigation 🕵️‍♂️

⁉️ **Hypothesis? What’s going on to create the latency?**

- Server-side rendering?

To test, we can switch to client-side rendering!

Investigation 🕵️‍♂️

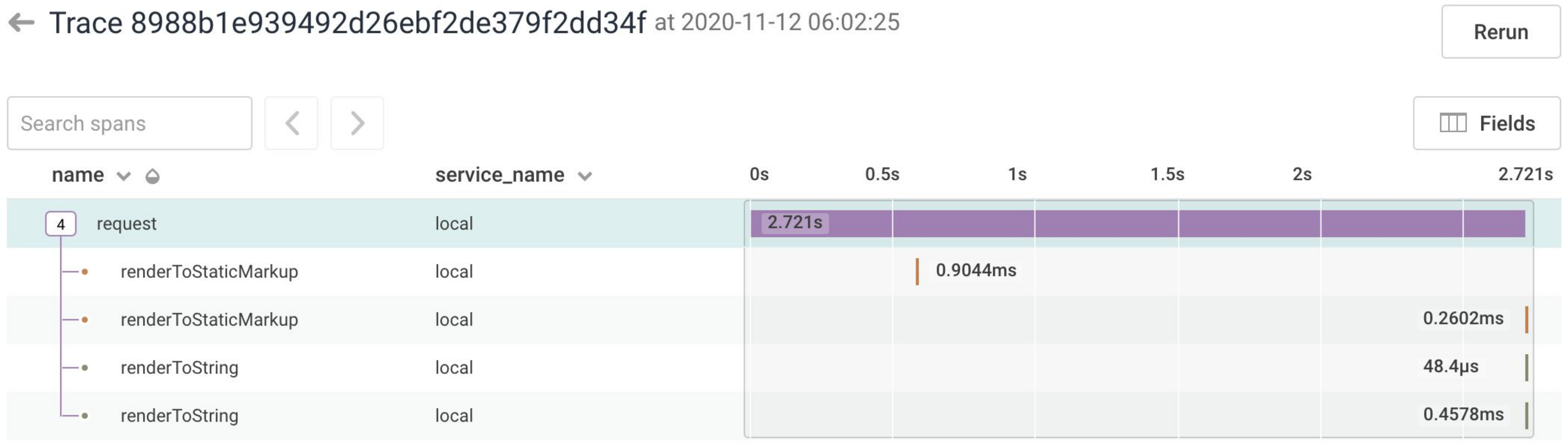
~~⁉️~~ **Hypothesis? What’s going on to create the latency?**

Server-side rendering!! ☠️

… but why?

Investigation 🕵️

Can we learn anything by looking at a trace?



Trace 8988b1e939492d26ebf2de379f2dd34f at 2020-11-12 06:02:25

name	service_name	duration
request (4)	local	2.721s
renderToStaticMarkup	local	0.9044ms
renderToStaticMarkup	local	0.2602ms
renderToString	local	48.4μs
renderToString	local	0.4578ms

Investigation 🕵️‍♂️

~~⁉️~~ **Hypothesis? What’s going on to create the latency?**

To do server-side rendering with React, we draw our React components on the server. This may calls `useQuery`, which (in the browser) makes an HTTP request to the server.

On the server, how do we make HTTP requests to the server?

Investigation 🕵️

⁉️ **Hypothesis? What's going on to create the latency?**

On the server, how do we make HTTP requests to the server?

```
export function renderApp(req: Request, res: Response) {
 const apolloClient = new ApolloClient({
   ssrMode: true,
   link: new HttpLink({
     uri: `http://127.0.0.1:${Config.appserverPort}/graphql`,
     credentials: 'same-origin',
...
```

Investigation 🕵️‍♂️

⁉️ **Hypothesis? What’s going on to create the latency?**

It is *much* too costly for the server to create bona-fide HTTP requests to itself.

If the result of making a GraphQL HTTP request to the server is calling functions in `api.ts`*, why can’t the server-side rendering just call into those functions directly?*

Investigation 🕵️‍♂️

💡 Fix! Tell Apollo to invoke the executable GraphQL schema directly.

```
export function renderApp(req: Request, res: Response, schema: any) {
 const apolloClient = new ApolloClient({
   ssrMode: true,
   link: new SchemaLink({ schema }),
   cache: new InMemoryCache(),
 })

...
```

Investigation 🕵️‍♂️

Let’s use the k6 load testing tool to see which routes / graphql operations don’t work well under heavy load.

```
export default function () {
 http.post(
   'http://localhost:3000/graphql',
   '{"operationName":"AnswerSurveyQuestion","variables":{"input":{"answer":"🤗","questionId":1}},"query":"mutation
AnswerSurveyQuestion($input: SurveyInput!) {\\n  answerSurvey(input: $input)\\n}\\n"}',
   {
     headers: {
       'Content-Type': 'application/json',
     },
   }
 )
}
```

Investigation 🕵️‍♂️

Anecdotally, I know the Survey app isn't scalable. It didn't work super well on the first day of class!

I can reproduce slowness locally without a load test just by answering a single survey question once I have a small number of rows in the database.

⁉️ **What's going on now?**

Investigation 🕵️‍♂️

⁉️ **Hypothesis? What's going on to create the latency?**

- Connecting to (or querying) the DB?
- Too many computations? Not enough RAM?

Aside: type systems

You are creating multiple type systems in your application:

- `schema.graphql` defines the ***shape of your API***. It is an interface implemented on the server by `api.ts`.
- MySQL schema defines the ***shape of your tables***. It is created for you by TypeORM (unless you manually modify the db via migrations).
- TypeORM entities define the ***shape of your objects stored & read from DB***.

Aside: type systems

You are creating multiple type systems in your application:

- `schema.graphql` defines the ***shape of your API***. It is an interface implemented on the server by `api.ts`.
- MySQL schema defines the ***shape of your tables***. It is created for you by TypeORM (unless you manually modify the db via migrations).
- TypeORM entities define the ***shape of your objects stored & read from DB***.

⚠️ **They do not always play nicely together!**

Aside: type systems

Recall the benefits of GraphQL:

- Strictly defined data
- Codegen (`npm run gen`) can create types for you 🎉
- Fewer round trips to server
- No tight coupling between frontend and backend
- Non-verbose API definition
- No over- or under-fetching

Aside: type systems

Recall the benefits of GraphQL:

- Strictly defined data
- Codegen (`npm run gen`) can create types for you 🎉
- Fewer round trips to server
- **No tight coupling between frontend and backend ****
- **Non-verbose API definition ****
- **No over- or under-fetching ****

Unfortunately, there's no free lunch. 😭

Investigation 🕵️‍♂️

⁉️ **Hypothesis? What's going on to create the latency?**

- Connecting to (or querying) the DB?
- Too many computations? Memory overload?
- Poor API design / implementation? Tight coupling? n+1?

Investigation 🕵️‍♂️

⁉️ **Hypothesis? What’s going on to create the latency?**

- Connecting to (or querying) the DB?

MySQL is designed to work with high concurrency. Why is this important?

Investigation 🕵️‍♂️

~~⁉️~~ **Hypothesis? What’s going on to create the latency?**

- Connecting to (or querying) the DB?

MySQL is designed to work with high concurrency. Why is this important?

- One query may be reading data from disk; another query can start processing
- Multi-core machines are mainstream

Investigation 🕵️‍♂️

⁉️ **Hypothesis? What's going on to create the latency?**

- Connecting to (or querying) the DB?

Your appserver connects to MySQL, but that doesn't mean it can execute queries in parallel! A single connection to MySQL can do one query at a time. If you want to do more than one query at a time, you need to have multiple connections open to the database.

Investigation 🕵️‍♂️

⁉️ **Hypothesis? What's going on to create the latency?**

- Connecting to (or querying) the DB?

Creating a database connection has some overhead associated with it. Given how frequently we query the database, we would ***not*** want to have to open a new connection for each query. ⁉️ **Why?**

Investigation 🕵️

⁉️ **Hypothesis? What’s going on to create the latency?**

- Connecting to (or querying) the DB?

Creating a database connection has some overhead associated with it. Given how frequently we query the database, we would ***not*** want to have to open a new connection for each query. ⁉️ **Why?**

- What if we open too many connections?
- What if a query finishes on a connection? Can we reuse the connection?

Investigation 🕵️‍♂️

⁉️ **Hypothesis? What’s going on to create the latency?**

- Connecting to (or querying) the DB?

```
export async function initORM() {
 return await createConnection({
    ...
    extra: {
      connectionLimit: 5, // this number is small
    },
 })
}
```

Investigation 🕵️‍♂️

⁉️ **Hypothesis? What’s going on to create the latency?**

- Connecting to (or querying) the DB?

ℹ️ Make sure your database can support the number of connections that you will attempt to open.

```
SHOW VARIABLES LIKE 'max_connections';

SET GLOBAL max_connections = 1024;
```

Investigation 🕵️

⁉️ **Hypothesis? What's going on to create the latency?**

- Too many computations? Not enough RAM?

Usually this is easy to see, but harder to identify the root cause of.

- CPU: look for trace spans that are long, events with high `duration_ms` or `SUM(duration_ms)`
- RAM: look for long lists, too much data being read out of your database

Investigation 🕵️‍♂️

~~⁉️~~ **Hypothesis? What's going on to create the latency?**

- Poor API design / implementation? Tight coupling? n+1?

Conflating our TypeORM entities (types) with our GraphQL types is tight coupling!
When should we load relations?

Investigation 🕵️‍♂️

⁉️ **Hypothesis? What’s going on to create the latency?**

- Poor API design / implementation? Tight coupling? n+1?

Conflating our TypeORM entities (types) with our GraphQL types is tight coupling!
When should we load relations?

- Eagerly?
- Only when requested by the client?

Investigation 🕵️‍♂️

⁉️ **Hypothesis? What's going on to create the latency?**

- Poor API design / implementation? Tight coupling? n+1?

The way GraphQL works, you have a bunch of **resolver** (functions). They have to return scalar values (primitives), objects, or Promise of either.

Any fields you put on your objects are visible to the GraphQL execution engine.

Investigation 🕵️‍♂️

⁉️ **Hypothesis? What’s going on to create the latency?**

- Poor API design / implementation? Tight coupling? n+1?

We can decouple the resolver functions for fields on our GraphQL schema from the TypeORM objects returned by resolvers. This is good hygiene for any relations that might not need to be loaded, because the GraphQL machinery can decide to call the resolver function if and only if the query requests that particular data.

Investigation 🕵️‍♂️

⁉️ **Hypothesis? What’s going on to create the latency?**

- Poor API design / implementation? Tight coupling? n+1?

Decoupling resolver functions can lead to the dreaded n+1 problem:

- Your GraphQL client requests a field on a query that returns a list (of objects)
- For each object returned, you want an edge (relation)
- Each edge calls a resolver function that requests a single row from the database
- For a list of n edges, you make n SQL calls to fetch n rows

Investigation 🕵️‍♂️

⁉️ **Hypothesis? What's going on to create the latency?**

- Poor API design / implementation? Tight coupling? n+1?

At scale you typically want to use batching and caching *liberally* to avoid unnecessary DB lookups, especially within a single request. The **dataloader** library provides automatic batching and caching for your resolver functions.