Query Languages

Spring 2025

Query languages for databases

Relational databases: SQL is the main query language for relational databases

- a single "entry point" into the DB
- text-based, programmatic — essentially you send a "program" to the DB
- DB executes the "program", returns rows of data as a result
- simple generic interface

NoSQL databases: not as fancy a query language

- KV dbs are essentially "lookup by key"
- Document databases have a more interesting JSON-centric query languages

Today…

Let's explore a bit the world of querying, talking about two query languages that arise in conversation, one recent, one old

- GraphQL
- Datalog

GraphQL

GraphQL

Need to talk about it because there's a lot of confusion out there

(What Riccardo isn't saying:
he *was confused about it until a colleague cleared things up some)*

GraphQL is *not* about databases

- it's not even about graph databases
- though it original came out from the graph database world
- if you want to dig into graph databases, go right ahead (Neo4j is a good start)

It's about providing an endpoint to a web-based service

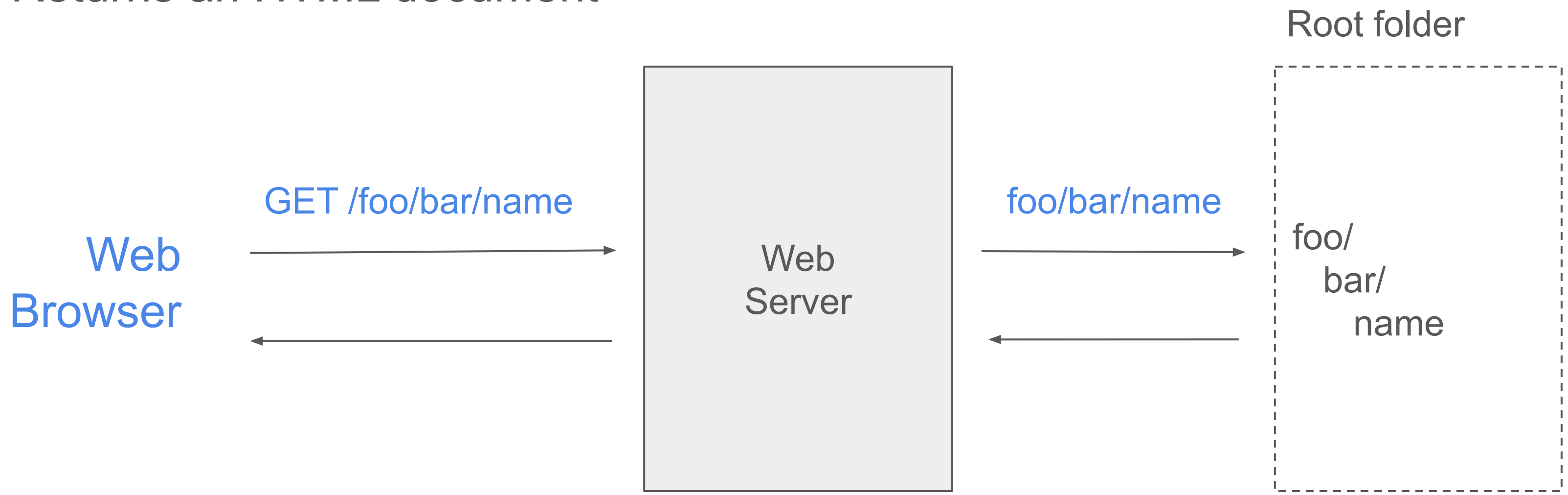
It's an alternative to a REST API

Web servers (1995)

Servers listening for HTTP requests on a machine at a port (80, 443)

Browser sends `GET /foo/bar/`

Server Returns an HTML document

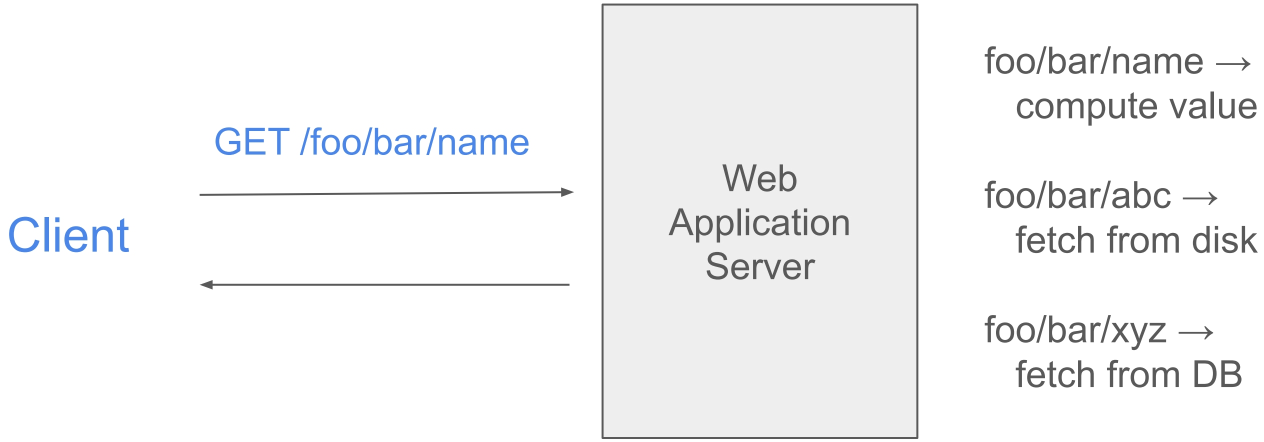


Web application servers (2025)

Servers listening for HTTP requests on a machine at a port (80, 443)

Client (browser, program) sends a `GET /foo/bar/`

Server returns an HTML document, a JSON object, …


GET /foo/bar/name
Client
Web Application Server
foo/bar/name → compute value
foo/bar/abc → fetch from disk
foo/bar/xyz → fetch from DB

Web app

Servers liste

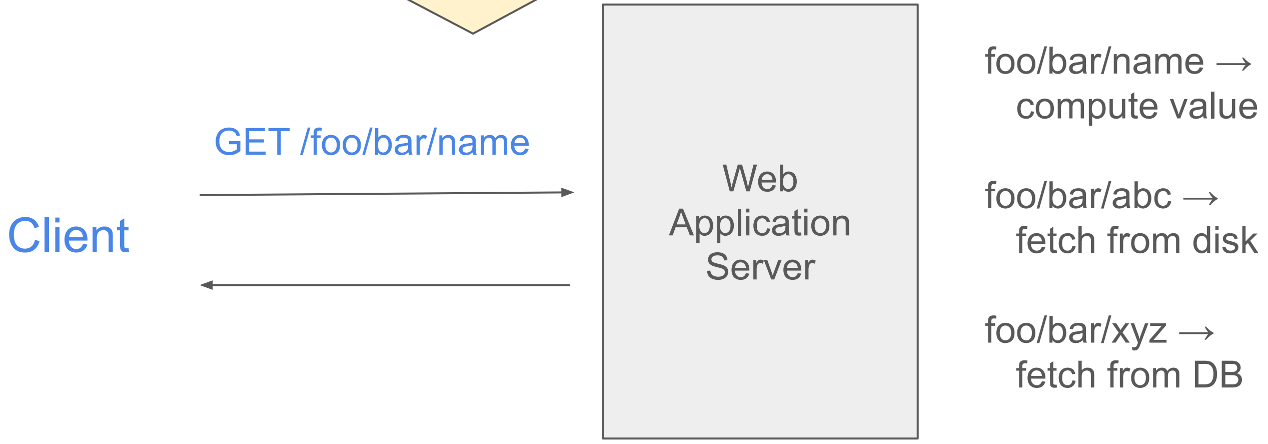
Client (brow

Server returns

You can think of these as *remote function calls*

You're calling a function on a server and getting a result back

The collection of functions on a server is often called an API


GET /foo/bar/name
Client
Web Application Server
foo/bar/name → compute value
foo/bar/abc → fetch from disk
foo/bar/xyz → fetch from DB

REST APIs

A common way to structure APIs that supply data (typically stored in a database)

- each endpoint correspond to a kind of resource
- resources may correspond to tables (really, entities in an ER model)
- but that's not a requirement (or even standard)

These get tricky when you need data across resources

- mostly reflects what you think clients will need to do
- either let them join things together at their end
- or try to model so that you can predict what they will need

New tables / resources → new endpoints

Example

Server to expose a simple blog / announcement system:

- users
- posts (a user creates a post)
- followers (a follower see all posts from a followed user)

Data in three tables:

Users, **Posts** (FK to user ID), **Followers** (FK to user ID)

A REST API would probably have three endpoints:

`GET /users/:id`	returns data for a user
`GET /users/:id/posts`	returns all posts for a user
`GET /users/:id/followers`	returns all followers for a user

Example

`GET /users/:id` returns

```
{
  "user": {
     "id": "...",
     "name": "...",
     "address": "..."
     …
  }
}
```

Fetch one row from table
Users
with ID `:id`

Example

`GET /users/:id/posts` returns

```
{
  "posts": [
    {
       "id": "...",
       "title": "...",
       "content": "..."
    },
    {
       "id": "...",
       "title": "...",
       "content": "..."
    },
    …
  ]
}
```

Fetch all rows from table
Posts
with user ID FK `:id`

Example

`GET /users/:id/followers` returns

```
{
  "followers": [
    {
       "id": "...",
       "name": "...",
       "address": "..."
    },
    {
       "id": "...",
       "name": "...",
       "address": "..."
    },
    …
  ]
}
```

Fetch alls row from table
Followers
with user ID FK `:id`

What about a direct SQL interface?

Generally a bad idea

```
    GET /sql/:q
```

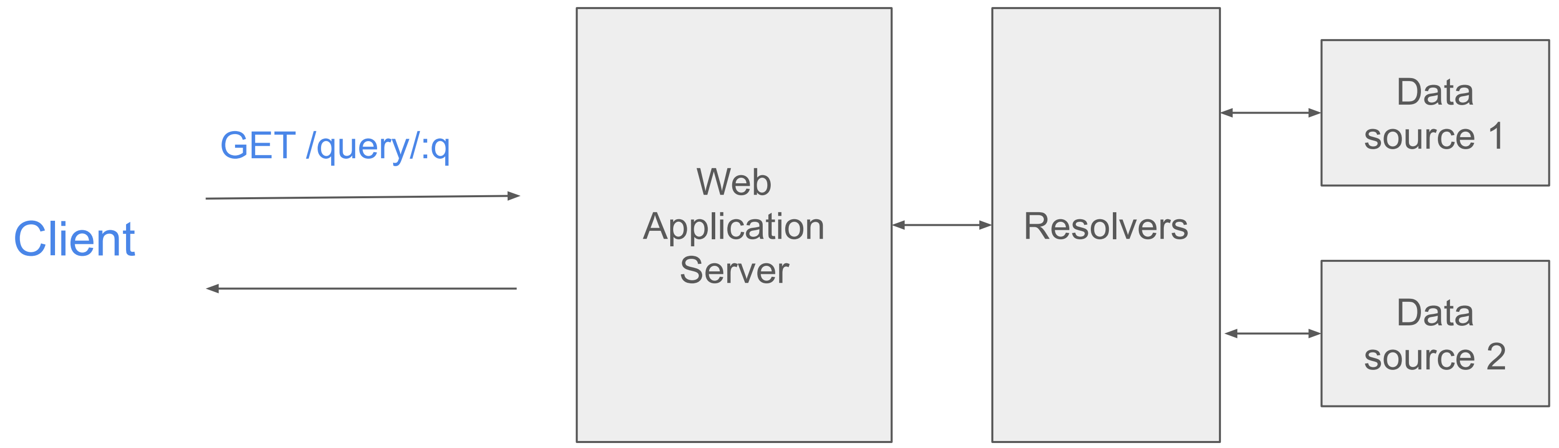
It forces API clients to know the structure of the data intimately

It constrains you if you ever want to change the structure of the data (new tables, split tables, etc)

Probably a security nightmare

GraphQL: take the idea, but introduce a different query language and use JSON

Architecture



GraphQL philosophy

GraphQL is really just a syntax for JSON-based queries:

```
query {
   User(id: …) {
      name
      posts {
         title
      }
      followers (last: 3) {
         name
      }
   }
}
```

Resolvers

Turning the query into a bunch of underlying calls to the various tables holding the data is the task of the *resolvers*

Resolvers convert the query into underlying SQL calls — if the data is in a relational database

GraphQL libraries of resolvers

- need to manually orchestrate resolvers: the posts of a users should be accessed as the `post` field of the `user` object, and merged by fetching the posts of the user from table **Posts**, etc

It's not a groundbreaking idea, but it's been packaged in a way that resonates with developers

You *can* think of it as a JSON-representation of a SQL query where you control the conversion

Datalog

Datalog

We know SQL as a way to query relational data

- formal model of SQL is known as relational algebra
- defined over relations as sets of tuples

Datalog is an alternative coming from deductive database theory

- useful when you have hierarchical (recursive) data
- a different approach to querying

A variant of Prolog, a *fifth generation* language (long story…)

org

employee	manager
Gautam	Mike
Wadah	Gautam
Kevin	Gautam
Van	Mike
Riccardo	Gautam
Riyaz	Riccardo
Jonathan	Wadah
Sneha	Riccardo

Datalog queries

A Datalog query:

- find for all tuples in a relation that match a certain template

```
org(X, riccardo).
```

- returns all tuples with *riccardo* in the second column
- i.e., all of Riccardo's direct reports
- capital letters are *variables*

```
org(X, _).
```

- return all tuples of employees that report to another

Rules

You can create intermediate (or derived) relations using deductive rules

The relation *Employees who are managers:*

```
    manager(X) :- org(Z, X).
```

X is a manager if there is some Z such that Z is a direct report of X

The relation *Employees that are on the same team:*

```
    teammate(X, Y) :- org(X, Z), org(Y, Z).
```

X and Y are teammates if there is a Z such that X and Y are direct reports of Z

Recursive rules

The relation *X reports to Y*:

```
    reports(X, Y) :- org(X, Y).
    reports(X, Y) :- org(X, Z), reports(Z, Y).
```

X reports to Y if X is a direct report of Y, or X reports to some Z that reports to Y

lead

employee	**team**
Gautam	Product
Wadah	Search
Kevin	Platform
Mike	Technology
Riccardo	Data

The relation *X is in group T*:

```
group(X, T) :- lead(X, T).
group(X, T) :- reports(X, Z), lead(Z, T).
```

Semantics

The meaning of a relation is the set of all tuples in the relation

- for a directly given relation, it's easy to figure out the set
- for a relation given by a (possibly recursive) rule, it's more interesting

As a set of tuples, *reports* is defined by:

$$\mathit{reports} = \mathit{org} \cup \{\, t \mid \exists z.\ t = (x, y) \wedge (x, z) \in \mathit{org} \wedge (z, y) \in \mathit{reports} \,\}$$

This is not a definition, it's an equation

- solve by finding the smallest solution (set) that satisfy the equation
- find a fixed point!

Negation

Negation is something that's kind of tricky in Datalog

- In SQL, you ask "is this tuple in the table" — it's a simple yes/no
- in Datalog, because of placeholders, you can theoretically ask "give me all tuples NOT in this relation" — how would anyone even implement that?

Negation in Datalog is *safe* and *stratified*:

- safe = a variable can only appear in a negated atom if it appears in a non-negated atom as well
- stratified = an ordering of relation such that a relation at stratum N can only refer to negated atoms over relations at stratum < N

Aggregation

Suppose the org relation takes an extra field holding the number of days worked

```
avgDaysWorkedDirectReports(X, AVG(<D>)) :- org(Y, X, D).
```

roughly equivalent to

```
SELECT manager, AVG(daysWorked) as avgDayWorked
FROM org
GROUP BY manager
```

When computing an aggregation, the underlying relation (here, Employee) must be fully known — same kind of stratification than for negation

For more details…

Datalog is a subset of Prolog, for which there is a lot of documentation

A good starting point is SWI-Prolog, a free Prolog environment

https://www.swi-prolog.org/

That's all, folks!