Day 4: Leveraging Relational Databases

Re:Coded - Server Side Development Workshop

Agenda

Today we will finish implementing our website backend!

First we will discuss how to handle the author name for posts and discuss normalization, primary and foreign keys, and joins.

Then we will handle upvotes and discuss indices, and different types of joins.

Finally, we will complete the site by implementing the Trending Posts query and learning about aggregate queries as well as revisiting our normalization discussion.

Table Relationships

When we say a database is “relational” we mean it has the ability to represent associations between rows in one table and rows in another.

It’s useful to look at a practical example to gain an intuitive understanding...

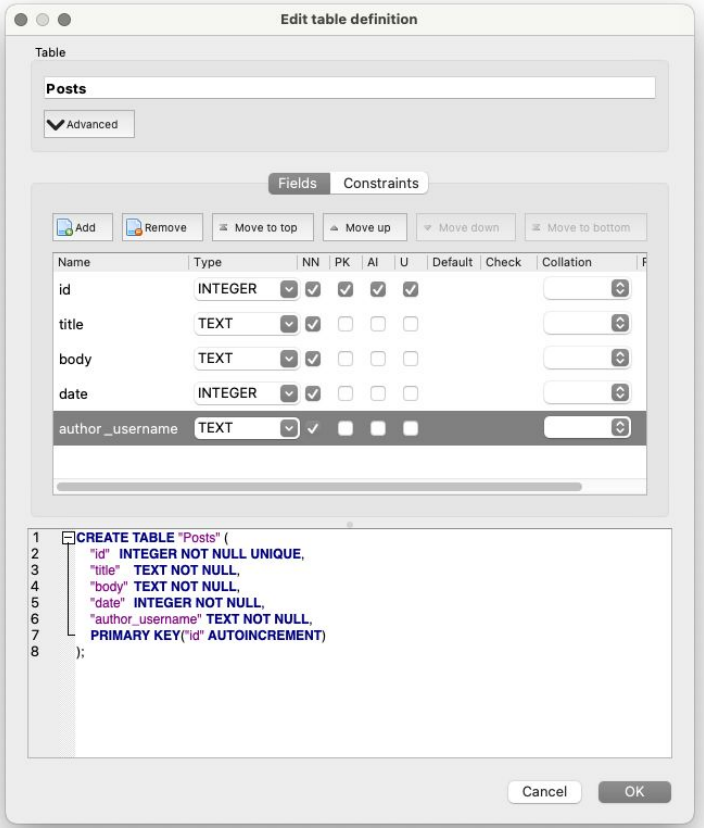
Denormalized "Author" Field

This is one example of how we could store the author username for a post.

A simple text field would do the job.

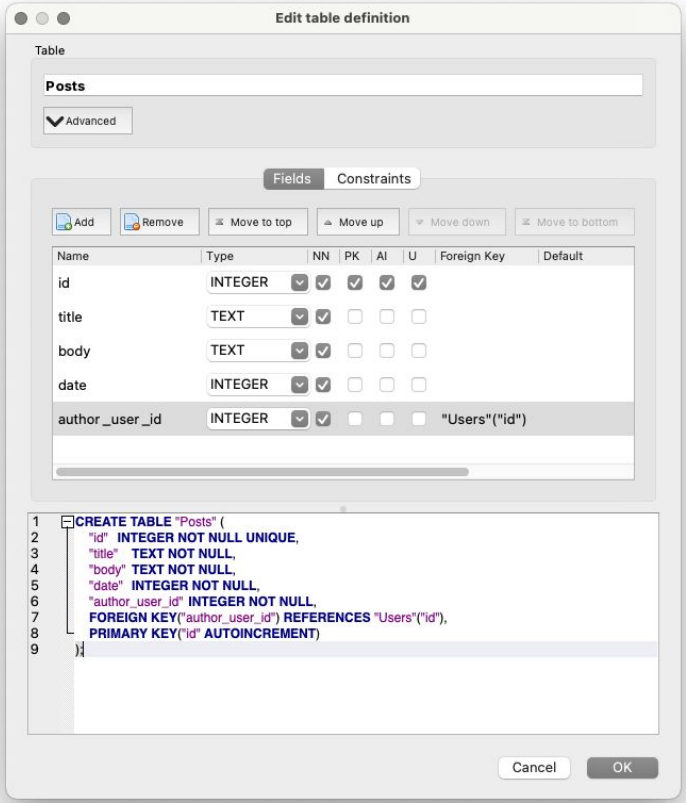
But what if the username gets updated?

Also, it's not space efficient.



Normalized "Author" Field

The “relational” alternative is to create a field whose value will be the primary key of the author’s user. In this way we don’t duplicate the information about the user (their username). Since there is a single source of truth for the username, we have eliminated the potential inconsistency.



Comparing Our Options...

Normalized

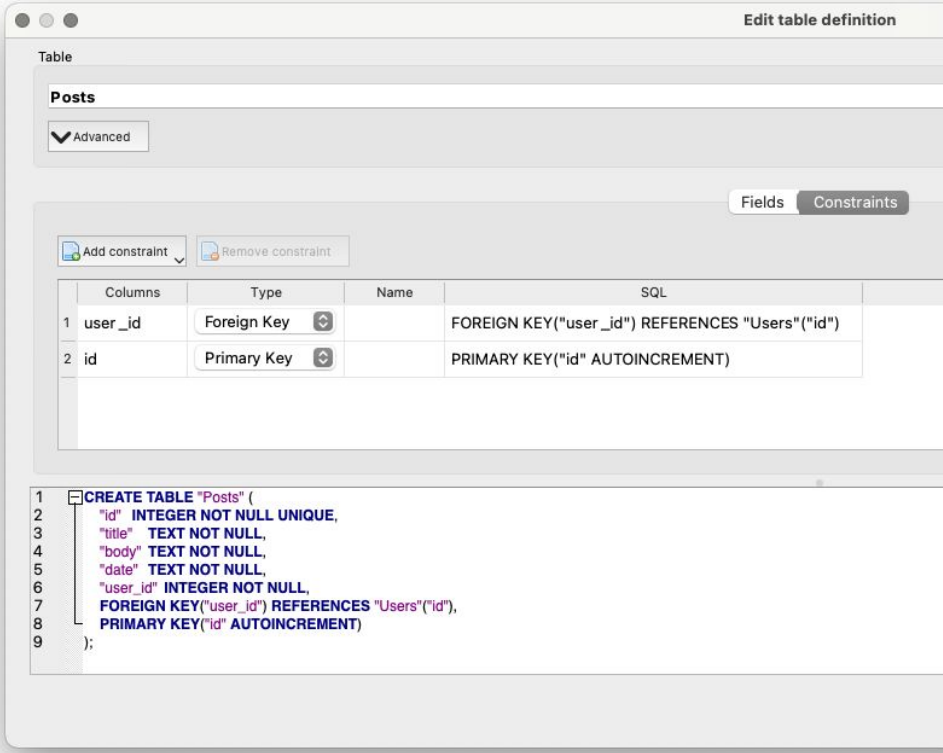
- **Slower** to **Read**
- **Faster** (free) to **Update**
- The database can guarantee that all foreign keys actually exist, that nullability constraints are satisfied, perform cascading deletes, etc.

Denormalized

- **Faster** to **Read**
- **Slower** (possibly very slow) to **Update**
- No data consistency guarantees from the database (your application is completely responsible for any consistency).

Primary Keys & Foreign Keys

By identifying these relationships to the database you enable the consistency guarantees we described.



Obtaining the Author's Username

```
SELECT
  Posts.*, Users.username AS author
FROM Posts
     INNER JOIN Users ON Posts.user_id = Users.id
```

How will we implement upvoting?

We have to tell users whether they have voted for a post, and prevent double voting.

Do we have options?

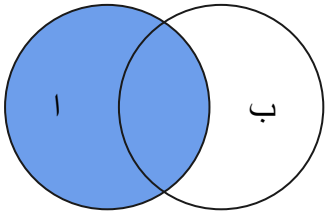
Describing Table Relationships

1:1 or “one to one”: Each row in the first table has a single corresponding row in the second table. Though, this description can also be used to describe the case where each row in a table has one or zero rows in the other table.

1:many or “one to many and many:1 or “many to one”: This is the most typical case. We would say that the Posts table has a many-to-one relationship with Users. Because many Posts can have the same Author (User).

We have seen how to handle this situation with authors - what about upvotes?

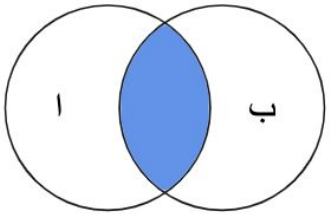
JOINs



FROM ا
LEFT OUTER JOIN ب ON ...

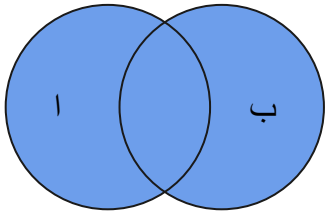
Typically for the situation where a matching row in ب is "optional". **Used very frequently.**

RIGHT OUTER JOIN exists as well, but is less frequently used.



FROM ا
INNER JOIN ب ON ...

When we only want the rows where the ON relationship matches rows in both ا and ب. **Used very frequently.**



FROM ا
OUTER JOIN ب

Rarely used. Essentially returns all combinations.

Ambiguous Table and Column Names

Joins introduce a new potential problem: what happens when we want to use the same table twice? Or when two tables have a column with the same name?

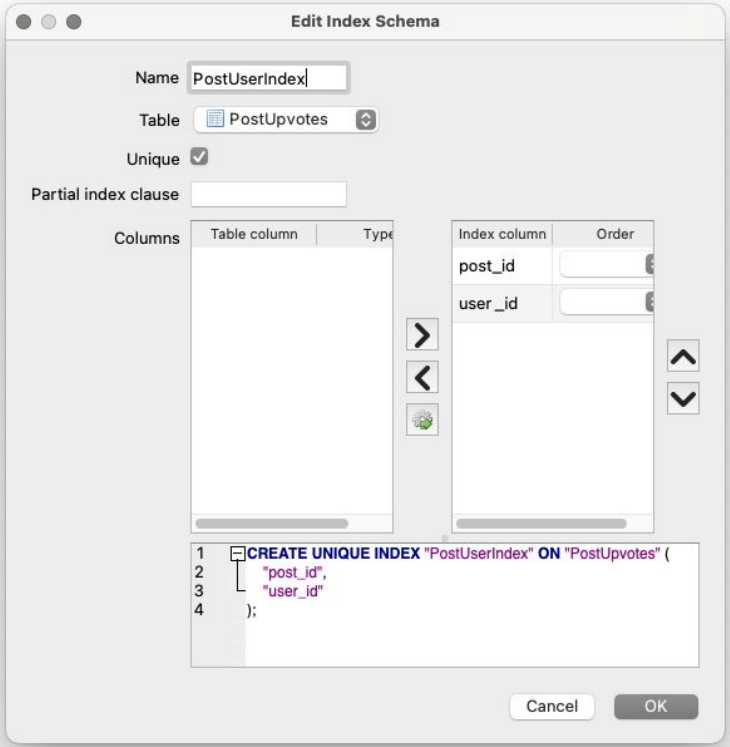
How will we implement upvoting?

Indexes

Indexes trade increased disk space for increased query speed.

They create virtual "views" of a table that are sorted by the fields specified in the index. They also power uniqueness constraints.

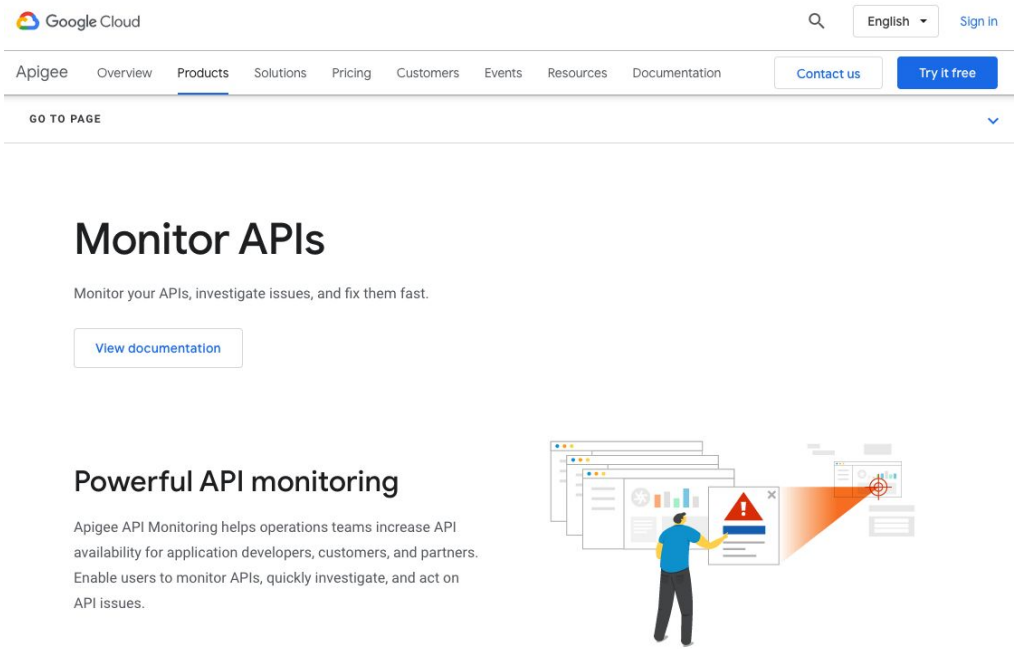
In the same way you can find a name in a phonebook faster if the phonebook is sorted, the database uses these sorted "views" to find rows relevant to a query faster.



Remember... to Measure

ماکو Index is better than having an unused index. Indexes are not free for the database to maintain. Unless you have a lot of experience, do not create indexes just because they "seem good" - test your indexes by averaging query times over many exemplar queries, for example - or, better yet, by observing changes in production executions. While not a great direct measurement of your queries, web service endpoint processing latency can be a useful indirect measurement, and many excellent services exist to automatically collect these types of analytics.

https://cloud.google.com/apigee/api-management/monitor-apis



Aggregating Data

We have solved the problem of tracking user votes on posts and not letting users double vote - but how can we implement our “Trending Posts” with this?

GROUP BY

One of SQL's tremendous powers is that it can aggregate values so easily. For example, you can:

SELECT SUM(total) FROM Orders
or:
SELECT AVG(total) FROM Orders

But what if you want to see these metrics "broken down"?

SELECT AVG(total) FROM Orders GROUP BY month

Common aggregate functions:

- SUM
- AVG
- COUNT
- MIN
- MAX

Consider when denormalization is useful

In the repository you will find two implementations of the "Trending" sort. Consider what benefits and drawbacks each has.

Does the denormalized one still guarantee consistency? (It doesn't - we can talk about transactions if you'd like to learn more!)

But perhaps being off by one or two votes if you're expecting millions doesn't make much difference?

See You Tomorrow!

We will talk about projects as well as open up for group hacking / Q&A.