When To Use SQL or NoSQL

SQLInSix Minutes · Follow
9 min read · Feb 1, 2024

In the video When To Use SQL and NoSQL, I compare a data example and how each would appear as an entity in a SQL and NoSQL database. In the past, I've recorded a series on SQL vs NoSQL databases that I still recommend:

- SQL vs NoSQL Part I

- SQL vs NoSQL Part II
- SQL vs NoSQL Part III

One quick misconception about NoSQL that I'll address quickly: NoSQL means not only SQL. With some NoSQL APIs, you may be able to write SQL against them. However, this may not be efficient relative to the data stored. Likewise if you define your architecture well when using NoSQL, you won't need to use a SQL syntax because you won't normalize your data.

Document Data Is Abundant

Initially NoSQL was treated with some skepticism — for instance Oracle wrote a white paper criticizing it only to release a NoSQL engine six months later. Some of this skepticism can still exist in some communities, I would advise caution around people discussing NoSQL as if it's never the appropriate solution.

Simply put: NoSQL represents and supports document data along with other data formats, but is most appropriately used for document data. Most data that exist today fall into this category. Normalization is a wasted exercise with this type of data because of how it's queried — you don't normalize data that is queried on the specific entity level that can vary significantly in structure. We don't want to be executing DDL statements regularly to update the schema as it changes.

If you've ever written a diary or journal, you actually have an understanding of document data because you would never dream of normalizing the data in a journal or its structure since both of these may vary significantly and over time. If you've ever seen different configuration file formats, you also have seem something similar — just because a configuration file leaves out a key doesn't mean that it is never or always needed. A fixed configuration by contrast is one where a table could be the source.

This is not to say SQL isn't useful. It's extremely useful for transactional systems. For an example, I've seen people try to use a document database for financial systems and they have poor performance because SQL is far superior. This is where understanding data is key: what data are you storing, how are you going to query that data, what are you going to use that data for, how often does the data structure

change or how flexible does the data structure need to be? I have extracted data from a database like MongoDB to put into a database like Oracle for deeper analysis, but that's only taking a portion of the documents in MongoDB. This latter point also highlights that there are numerous situations in which you'll use both technologies.

(Often when I speak about NoSQL engines to SQL types, I point out that a document database is like a file system. When you think of it that way, it makes more sense in terms of a flexible schema where you may want to further extract some data from the files for further analysis, as this is what SQL developers tend to understand from their experience. The good news is that NoSQL is no longer resisted as much as it was initially, so there's less pushback — it's proven that it's here to stay.)

Terminology

In a SQL table, we have a table within a database that has columns and rows of data. We can define a unique key called the primary key made up of one or more columns. If we normalize our data, we can have references to other tables through the use of foreign keys, which mean that a column in our table references another column key in another table. To explain this terminology so that we understand the hierarchy:

- A foreign key column links a column in one table to another table
- A row of data is the entity or the data value
- A column is an attribute or property of the entity
- A table holds all the entities that are grouped together in a fixed structure of columns
- Tables exist within a database

In MongoDB (a NoSQL database), we have a collection within a database that has documents. MongoDB does enforce a unique key if not defined by the user (the `_id` field) that cannot be duplicated without an error. Within a document, you may have subdocuments and these subdocuments are data related to the document itself — for an example, a list of addresses that a person has. To explain this terminology so

that we understand the hierarchy:

- A subdocument is a child document within a parent document that is defined by the parent's document
- A document is the entity or data value
- A field is an attribute or property of the entity
- A collection holds all the entities that are grouped together in a fixed structure of columns
- Collections exist within a database

When SQL or NoSQL

Regardless of what back-end we choose, we want to think of our data in terms of organization and how we'll query our data. One of my early mentors Adrian Luff once told me that object oriented developers tend to think of databases as a trash can: they throw data into the trash can and then need to go get that data and can't find it. Part of why this is involves how we create and organize our data model and also how we'll query our data. I discuss this observation in the video Tech Pow Wow: Databases As A Trash Can.

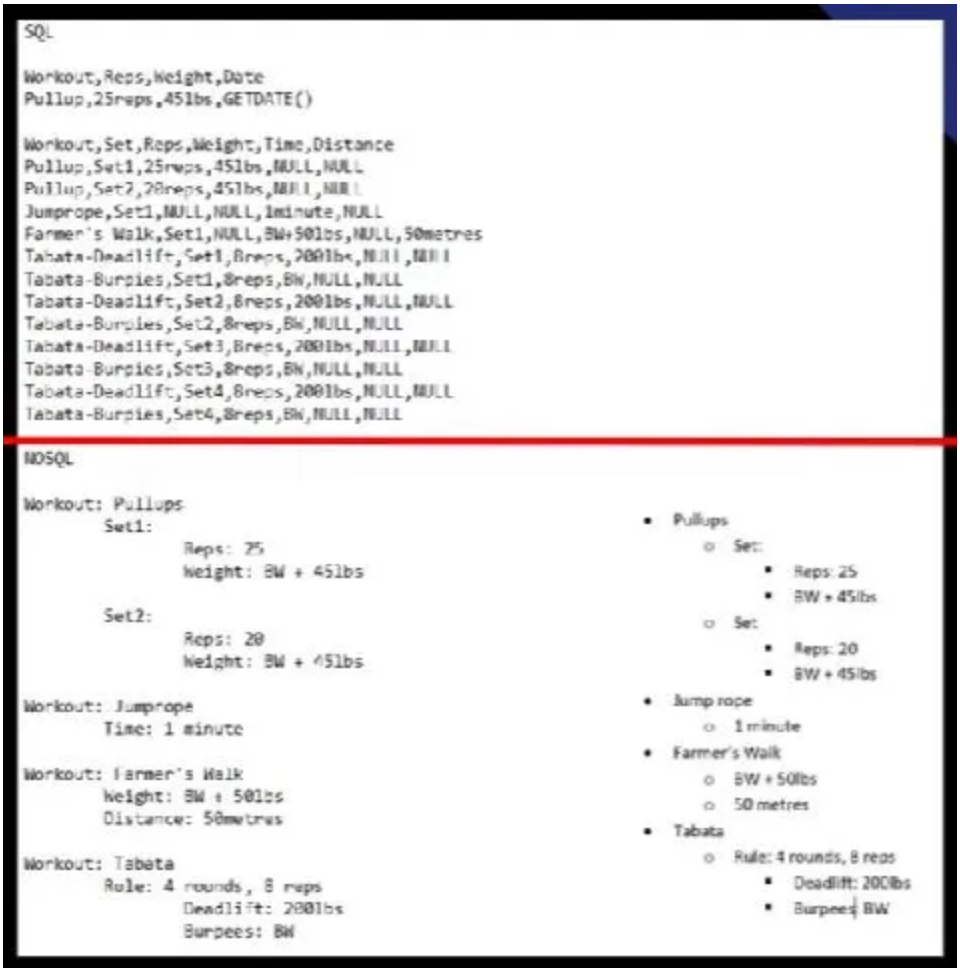
Some questions that I consider when I compare these different back-ends:

- How often will I be reading and writing the data?
- How often will the structure of the data change?
- How much of the data is unknown — meaning that more can be added later that I may be unaware of? As an example to this point, there is no such financial transaction as a addition or subtraction meaning that we know all the types of financial transactions that will exist — additions or subtractions. However, as we see in the below example, we may realize that we don't know every possible workout that can exist and this could complicate our schema or normalization design.
- What does the final product look like?

These make good starting questions to ask about SQL or NoSQL, as the answers to each will highlight the appropriate solution.

Sometimes, I may use NoSQL when most people would think that SQL would be superior because of the final product or the purpose. For an example, recently I completed a project where an architect chose SQL as a solution but this was a poor choice. The final product was a flattened data format that passed in fields to a machine learning model. The entire SQL project was essentially grabbing all the data and tying it together to create the equivalent of a document in MongoDB. However, this would have been better served by being a document in the first place. Who ultimately wins when the correct technology is chosen is the client, as 500 hours of work can become 150 hours of work.

Data Example



SQL vs NoSQL data example using fitness.

The above image shows our data example with a data entity stored in a SQL database, then stored in a NoSQL database. Using the structured data approach with SQL, we would attempt to define tables that have a fixed schema associated with various workout types. I note this because workout types can vary significantly if you're familiar with the data — for an example, a workout with weight and reps is only one type, similar to a jump rope circuit that is timed and must hit a number is also another type. The challenge with SQL is that you must define these workouts ahead of time and be prepared to continue to expand your schema as new workout types are added. To normalize our data with tables that have various workouts, each of these tables would have a foreign key that ties back to the workout table (ie: `WorkoutId` value of 1 that would be associated with `2017-01-01` as an example in the main table and that `WorkoutId` in the related table would be the type of workout with the result).

The join complexity would be enormous relative to a person's workout, as there may be 5–10 (or even more) combinations. In addition, workouts such as drop sets where the weight is constantly changing and a maximum (or minimum) repetition range must be met would be extremely challenging to track with a set schema using SQL. Of course, it's always possible that you get the person who sticks to the same workouts every year.

NoSQL stores semi-structured data. In our video's case, we're using MongoDB which stores BSON (binary JavaScript object notation). When we think about NoSQL from the start, we think about the data entity itself — for an example, we think about the entire workout of day `2017-01-01` as a whole. We're not going to query all the tables that could possibly have a foreign key relationship with that table because we will store the entire workout as an entity in a document. A document might have subdocuments, which in our example's case will be the type of workout (reps-weight splits, drop sets, jump rope circuits, sprints, etc). However, the key difference here is that we don't need joins to return all the related data from a normalized data process because we store by the entire entity.

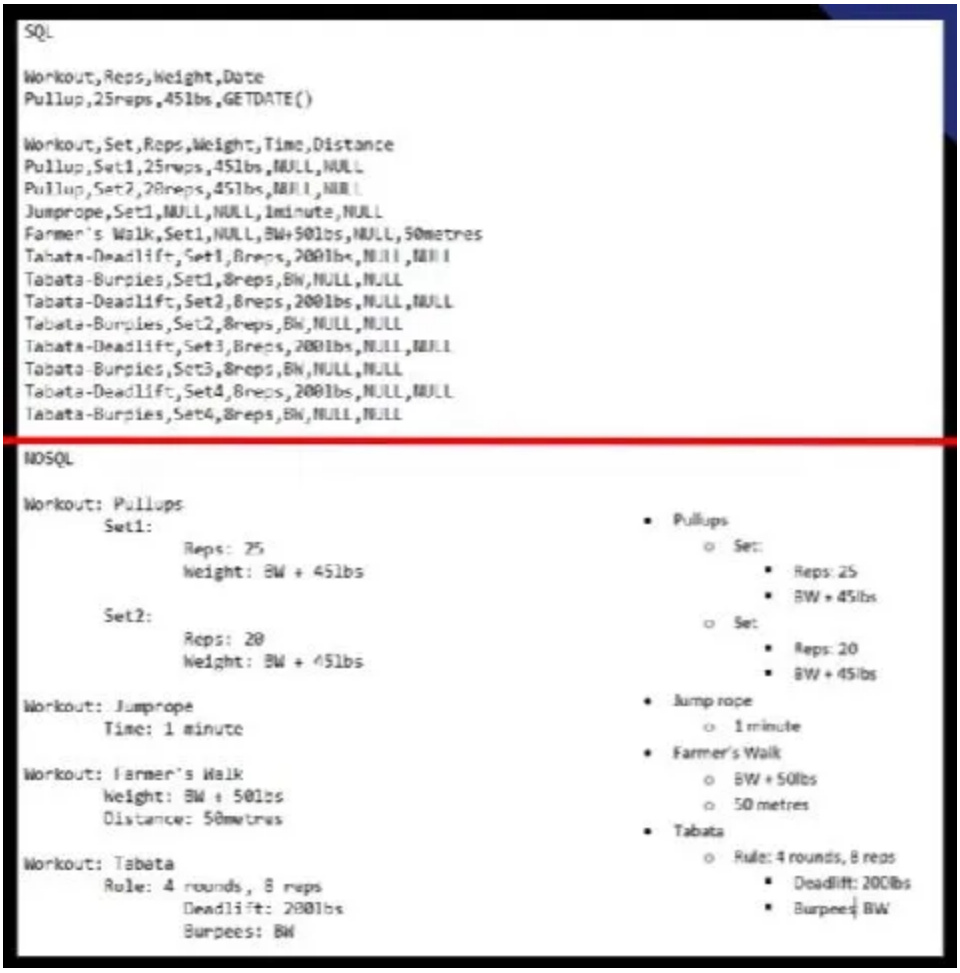
Reports Vs. Writes

In OLTP systems, we optimize for writes as much as possible — we want to get the

data and update them as fast as we can. Streaming is immediate, but luckily only covers writes without updates initially as we need the data. Think of a streaming and OLTP system like getting sensor data from cars (streaming insert), then transforming the data (OLTP update) before adding them (OLTP insert) to our database. This is a streaming plus OLTP combined scenario and is actually very common in the IoT space.

However, what about an example where we store a customer's mailing address? In my view, this is a terrible idea in a SQL database because the normalization approach to this lacks an understanding of business necessity. Very few people are updating their address on a daily basis, so optimizing for writes makes little sense. An address is going to be read more than written. NoSQL is a much better way to store addresses and address history because (1) we'll be reading it much more than writing a new address and (2) NoSQL makes it easy to store historical addresses in a subdocument.

Returning to our fitness example, we can also consider reporting data versus writing data. When would we update this? After writing it once, never. You may add a new workout, but you're never going to update a workout as once it's over, it's over. Thus fitness data will be read at a much higher rate than ever written. Therefore, we want to optimize for reads and we want to optimize for how we'd use a fitness application.



I like fitness in this example because it makes an extremely good data example of a "why" — what would we do with the data that we store? For some, comparing results would be the use-case and a SQL solution might be better for comparing values. For people who tend to have more dynamic workouts (they change up the style), this is less important. More than likely, they'll be querying the data to see the various types of workouts they did in the past. In reality, very few people will be even reading this data for comparison over the long run.

(Outside the scope of this article, but anyone with fitness experience knows that comparing lifts when you're 18 to 35 is dumb, so even if you were consistent enough to workout throughout that period of time — and most aren't — you wouldn't even be comparing strength at those ages. While this detail seems minor for this example, it does highlight the importance of *data nuance* in business, as these minor details tend to have huge cost savings when you know the why of how to store data, or even

if you should store data because of how data will be used).

Conclusion

As I highlighted nine years ago about NoSQL databases, they are here to stay because most data fall into the document category. This doesn't mean that we should overapply them in solutions where alternatives may be superior. We should consider the data situation and use the best tool to solve the problem.

When I consider whether they are the solution, the order that I think of involves data flexibility, read operations, and last of all, write operations. As we see with some examples, these helps us determine whether we've started with the best solution.

Note: all images in the post are either created through actual code runs or from Pixabay.

Sql NoSQL Mongodb

Data & Security

Follow

Written by SQLInSix Minutes

62 Followers · 0 Following

I speak and write about data and security. Given that I am speaking more on technical topics, I will be posting once a quarter going forward.

Responses (1)

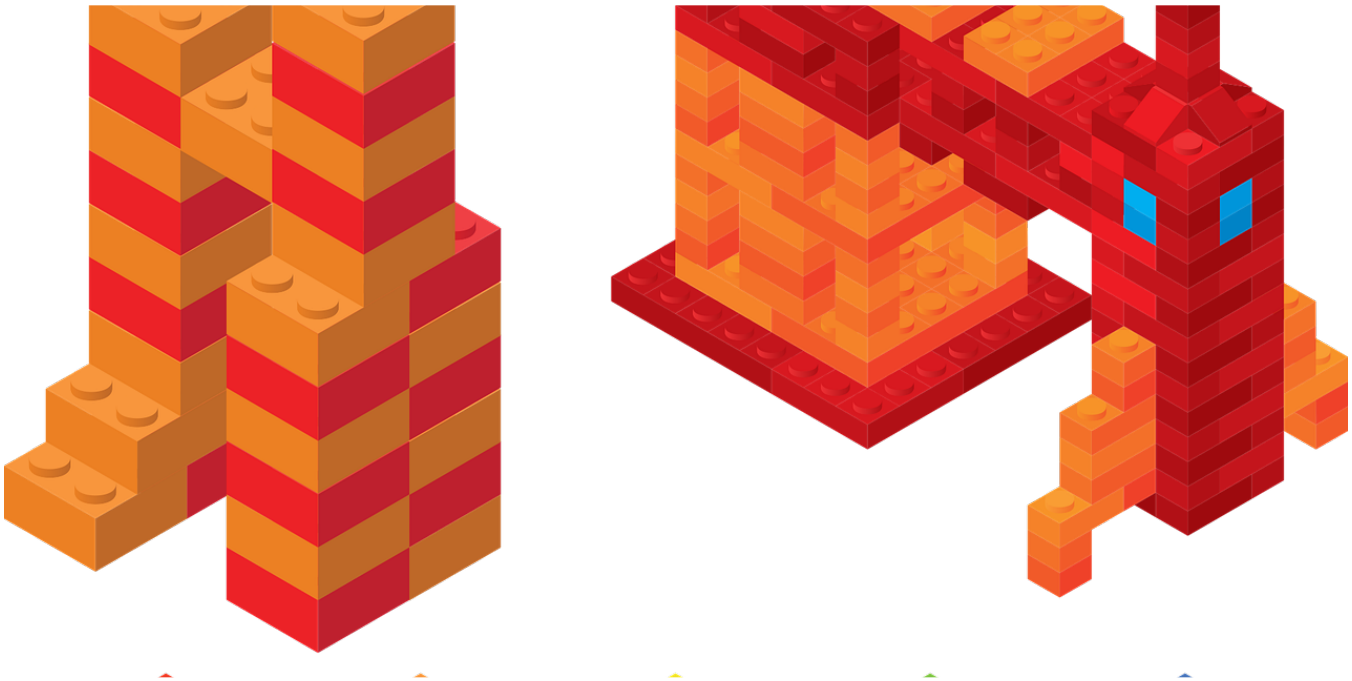
What are your thoughts? Respond

Rahul Kumar
21 days ago

https://medium.com/@rahul.kumar0/sql-vs-nosql-picking-the-right-database-for-your-data-drama-754fd2db8528?sk=v2%2F28add819-9737-4b72-b380-f8ec0a8727f1

Reply

More from SQLInSix Minutes



SQLInSix Minutes

Using the OPENJSON Function In SQL Server

In this post we look at using the OPENJSON function in SQL Server to parse some example JSON documents. As a quick note, I would not use...

Feb 14 53

SQLInSix Minutes

How To Write A Left Anti Join and Why

Previously, we've look at the join functionality of LEFT JOIN. If we recall from that lesson, we learned that a LEFT JOIN will join records...

Jul 1, 2021 5

SQLInSix Minutes

How To Compare 2 Tables In Different Databases

In the video, SQL Basics: Compare Tables In 2 Different Databases, we look at how to compare two tables using SQL. In the context of the...

Jan 27 44

SQLInSix Minutes

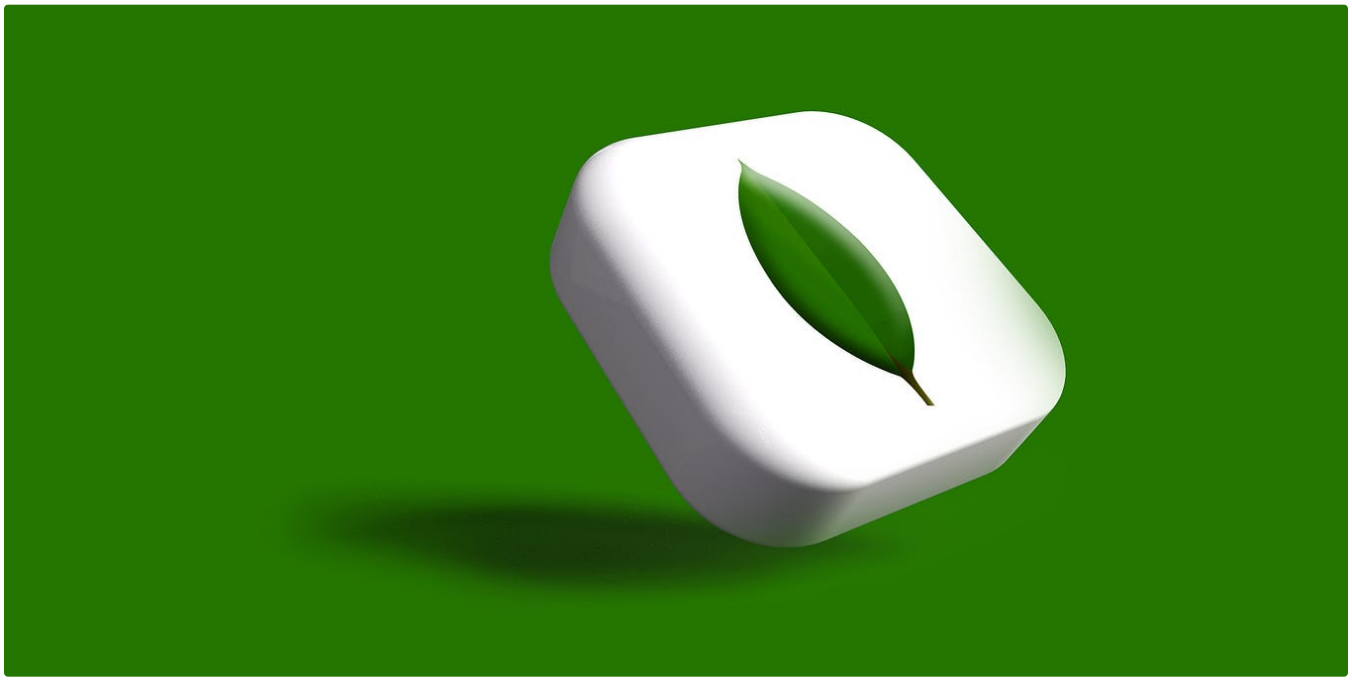
VNet Injection With Azure Databricks

One challenge that we can face with labor division involves two teams that must design the appropriate technical architecture with neither...

Apr 6 4

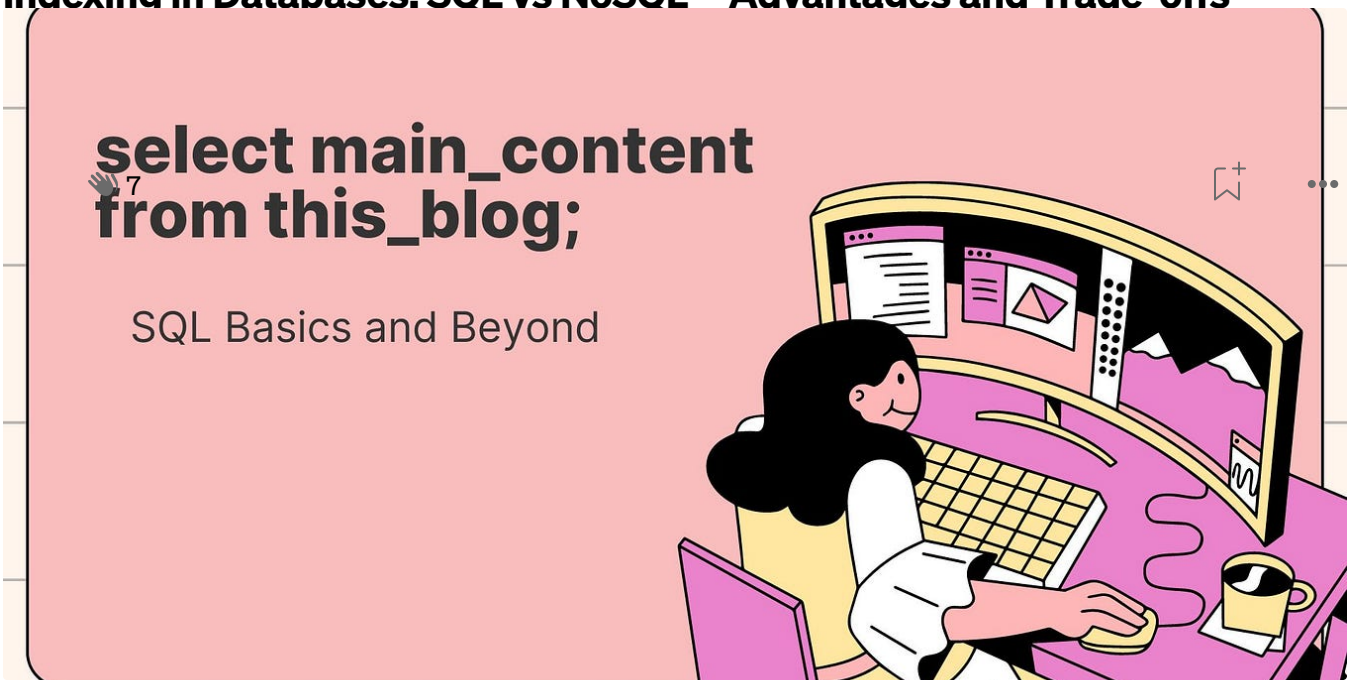
See all from SQLInSix Minutes

Recommended from Medium



Badri Paudel

Indexing in Databases: SQL vs NoSQL—Advantages and Trade-offs



In DevOps.dev by TechieTreeHugger

SQL Showdown: CTEs vs Subqueries vs Temp Tables vs Views

Boost your SQL Game !!

Oct 1 22

Lists

ChatGPT

21 stories · 899 saves

Natural Language Processing

1842 stories · 1466 saves

Back-of-the-Envelope Estimation

In Level Up Coding by Arslan Ahmad

Back-of-the-Envelope Estimations: A Comprehensive Guide for System Design Interviews

Learn the art of quick estimation in system design interviews with this comprehensive guide, packed with practical examples and tips for...

Apr 27, 2023 665

Manish Sonal

Systems Design Interview: Design a Ticket Master

Systems design is subjective and it evolves with time and requirement updates, there is nothing as such an ideal design. A good system...

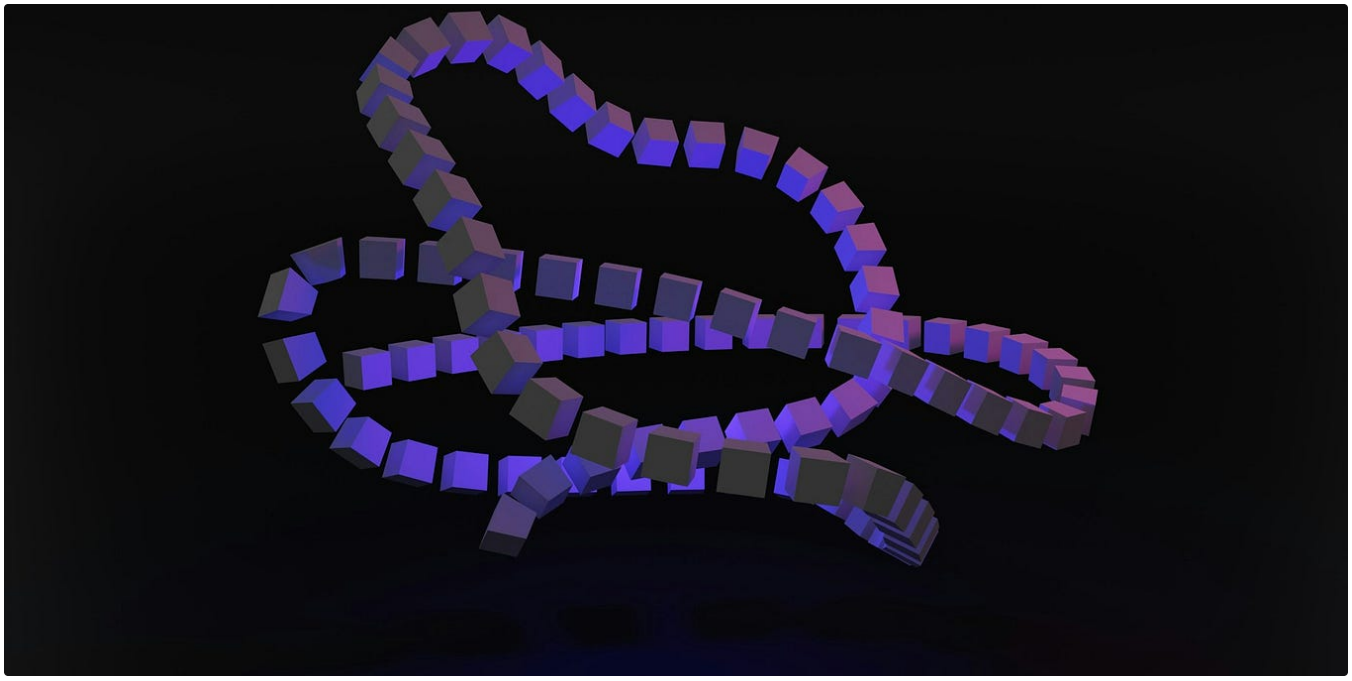
Sep 22 48

Meghamishra

Mastering MongoDB Aggregation Pipelines: A Guide for E-Commerce Data Analysis

When working with databases, especially with large datasets, we often need to analyze or process data in more complex ways than simple...

Oct 29



In Xeynergy Blog by Niraj Ranasinghe

Achieving Scalability with Sharded Database Architectures

In the digital world we live in today, applications work with enormous amounts of data. To maintain performance and scalability, especially...

Jun 20 16

See more recommendations