Database types

Comparing relational and document databases

CONTENT

Introduction

Over time, databases have been designed to accommodate many different usage patterns, organizational hierarchies, and consistency constraints. Two of the most enduring designs are relational databases and document databases.

In this guide, we'll take a look at these two database models to get a better idea of their relative strengths and weaknesses and evaluate what scenarios they're best suited for. We'll take a look at how they handle data structure, querying, and the upshot of using either within your projects.

How relational databases and document databases organize data

The data structure of relational databases is defined primarily through a hierarchy of related mechanisms: databases, tables, and columns.

Databases themselves act as a container for various tables and properties within the system. The database layer allows administrators to apply policy and attributes to sets of related data globally. They also serve as namespaces to limit the potential for naming collision for tables and other child elements.

Within these databases, the data's structure is defined by tables. Tables are made by declaring the names and properties of a set of columns and configuring table-wide attributes.

Each column specifies a data type which controls the shape of data that may be stored within it. Constraints can also be declared on columns and tables which allow you to impose additional requirements on what constitutes valid data for the field. The data itself is stored as rows within the table. Each record stores a single value for each of the table's columns.

To summarize, relational database management systems (RDMSs) group related tables and settings into databases. Each table defines a specific structure for the records it can hold by setting up a series of columns that have a data type and other properties. Records are added to tables as rows, with each row recording a value for each of the table's columns.

How document databases structure data

Document database management systems also structure their data through a hierarchy of related components, but with a different set of paradigms. Document databases typically use a system of databases, collections, and documents.

As with relational databases, document database systems use an overarching "database" abstraction to encapsulate related data to allow for global policy and namespacing. The database layer serves as a container to define wide-ranging properties, allow for cohesive access control, and to scope actions to the relevant context.

Within the database, a grouping called collections are used to bundle together individual documents. Collections are more loosely defined than tables (their relational counterpart) and mainly serve as a container and additional scoping mechanism.

Unlike relational databases, collections themselves do not define the fields or properties of the documents that they can store. Instead, the structure of each document is defined implicitly by the fields and data that it declares. Because the structure being a property of individual documents rather than a collection, the shape of the documents within a collection can vary widely. Conforming

Given the different data management philosophies that these two designs employ, it might not be surprising that there are some major differences in the amount of structure and flexibility afforded to users. In general, relational database systems tend to prioritize structure, predictability, and consistency while document databases prefer flexibility, responsiveness, and adaptiveness.

The case for structural rigidity

By using tables, relational databases configure the shape of their data ahead of time. Each record that is stored within a table must conform to the structure that the table implements without exception.

To change the structure of the data, the table structure itself must be altered and any existing records will need to be updated to match the new structure. This system makes structural changes relatively expensive as each piece of data already entered into the table requires an update. This can mean updating every record in the table, rebuilding indexes, and having to make decisions about the best way to backfill values that weren't recorded at their initial entry.

This cost makes it wise to think carefully about your data structure ahead of time, which can be intimidating. However, it is important to keep in mind that this method provides a good deal of reassurance and safety in terms of data integrity.

The data stored in a table will always be homogeneous to the extent required by the table definition. It is a mechanism of enforcement that can help you maintain well-ordered data that can be reasoned about without introspecting the individual properties of each data item.

The table structure offers guarantees about your data that isn't possible without a consensus around what the data should look like. This can help you avoid entire classes of problems in terms of data consistency and coherence, especially over time as the application logic that interfaces with the database evolves.

The case for flexibility

On the other side of things, document databases define the structure of each individual document separately. The structure is a characteristic that the document itself defines rather than an *external* structure the record must conform to.

This gives you a great deal of flexibility in a number of different areas. You can change the data you want to record for individual records on the fly, delay or skip backfilling previously saved documents, or store documents with very different structure within the same collection without requiring to set

The flexibility provided by the document model encourages the iteration and evolution of your storage logic. It is important to keep in mind, however, that the software itself is unlikely to be able to provide you with as many guarantees about your data as you make changes. If there is no agreed upon standard for the shape of the data collections hold, then it is up to you as a developer to enforce consistency and to modify documents where appropriate to keep your data in a well-understood state.

Normalization, joins, and data cohabitation

One of the most useful features of the relational database model is the concept of joins. Joins are queries that allow you to stitch together data held in different tables to allow you to work with them as a single unit. While it might make sense from an efficiency, logical, or consistency standpoint to store certain pieces of data in distinct tables, often you'll want to retrieve data or otherwise operate across those boundaries.

Normalization is a table design philosophy that encourages you to organize data in a way that guarantees consistency of data across table boundaries. Normalization is in some ways a requirement for joins while also being one of the primary reason why joins are necessary within the relational model.

You can think of normalization as a strategy for maintaining data consistency by logically separating data according to how reusable its discrete parts are. For instance, a record that represents a unique customer will be used very differently than a record that represents one of the customer's individual orders. Normalization encourages you to separate these types of information into their own tables, while joins provide you with the mechanism required to combine them when needed.

Joins require consistently structured data with fields that can be mapped to one another to match individual records. Because the document model doesn't enforce a set structure for records, a join is a more difficult operation to conceive of within that paradigm. With that being said, it's important to note that:

- similar functionality can be achieved in many document database systems
- the model itself encourages less diffuse data storage

What this means in practice is that the document structure does not have the same pressures driving towards normalization that relational databases have. Records can and often do contain nested information that would usually be separated in a relational model, allowing you to retrieve related information in a single document rather than stitching together multiple entities. Even so, aggregation operators can provide some of the same basic functionality as joins within the document

modify database structures, and manage the general database environment. SQL has been around since the 1970s, has many different iterations and implementations, and is generally regarded as the standard interface for databases that follow a table-based structure.

SQL has many proponents and detractors and can be a polarizing subject, in part due to its long history, inconsistent standards, and different ergonomics compared to traditional programming languages. SQL is fairly expressive and flexible in what it can access and retrieve, but it can be cumbersome to deal with programmatically because its grammar and the structure of its statements can be difficult to parse and construct. Furthermore, it doesn't follow the patterns laid out by many other languages because, as its name suggests, it was conceived primarily as a command and querying language rather than a general purpose language for programming with data.

For relational databases, SQL is often the primary means of interfacing with the database system. With document databases, the record structure isn't aligned with the assumptions inherent within SQL, so new querying languages are required for interacting with the system and entering, querying, and modifying data. Some languages take a good deal of inspiration from SQL while others implement entirely new languages in an attempt to escape from some of SQL's more frustrating warts.

In many document databases, the querying language has been designed to follow access patterns that may be more familiar to application developers. They often implement an API-like interface that emulates the tooling that developers use for JSON-like data management so that they can reuse existing patterns. While relational databases tend to converge around the SQL standard, document databases and other NoSQL databases may have very different interfaces from one another. Often the querying language that's developed alongside the actual database is one of the points of differentiation between different document database solutions.

Scaling beyond a single database

Most traditional relational databases were initially designed in an era that assumed a fairly simple infrastructure environment. In many cases, vertical scaling, or scaling the performance of the database by adding additional resources to a single host, was the simplest strategy and one that could usually satisfy any new requirements.

Horizontal scaling was also possible either by replicating databases to multiple followers or by sharding data: splitting up databases between several hosts so that each is responsible for a subset of the complete data. These strategies allow for some flexibility beyond just managing the resources of a single host and also provided additional benefits like increased availability and failure mitigation.

Newer relational databases have been developed to help address these issues, so there has been significant progress in this area. However, some of the difficulty in scaling is inherent in the relational model itself. When constraints and consistency requirements are set across large portions of the data, some level of coordination overhead must exist to manage the state data across various hosts. In some important ways, the relational model seems to require a centralized management architecture.

Document databases, on the other hand, are able to avoid many of these shortcomings due to the way that they data is structured within their systems. By collocating related data together in a single document, the coordination between different hosts can be minimized. Sharding datasets is a much more common strategy in document databases. This is due to the fact that document-based operations typically don't require as much coordination since many actions target individual records.

Because fewer constraints and links exist between individual documents and collections within document databases, coordination is often easier and operations tend to be more self-contained. This allows document database providers to prioritize performance and availability, where relational databases are often forced to make concessions in the name of consistency. This is a trade-off that can have many implications for the safety of your data and how well your systems can handle outages and network partitions. The major difference ends up being that document databases tend to have much more flexibility in tuning the level of consistency versus performance and availability while relational databases often require consistency to always be the first priority.

Conclusion

Relational and document databases have, in some ways, very different approaches to organizing data and making it manageable and accessible. Their basic designs have far-reaching implications for the types of applications they are suitable for and what challenges you are likely to see when working with them.

While relational databases and document databases can both be used in some scenarios, the priorities and development practices of your project often align better with one system or another. It is worthwhile to carefully evaluate both options when possible to understand the trade-offs you may be making and which system offers the features you really hold important for your work.

Justin Ellingwood

Justin has been writing about databases, Linux, infrastructure, and developer tools since 2013. He currently lives in Berlin with his wife and two rabbits. He doesn't usually have to write in the third person, which is a relief for all parties involved.

Previous
← In vivo: information ecosystems

Next
The benefits of PostgreSQL →

Edit this page on GitHub