Why the Document Data Model

How JSON Documents Help Developers Innovate Faster

mongoDB

Table of Contents

Introduction

Relational databases have a long-standing position in most organizations. Based on a tabular data model composed of rows and columns, relational databases – designed back in the 1970s and popularized in the 80s – became the default way of managing data.

However as developers build applications for today's digital economy, they are facing demands that stretch the limits of what's possible with relational systems. These demands include:

- Accelerating developer productivity to keep pace with a constant stream of ideas from the business, compressing application release cycles from months to days and weeks.
- Storing and querying rapidly changing data that arrives in all shapes and sizes – structured, semi-structured, and polymorphic – where defining a flat, tabular schema in advance is not practical.
- Distributing data across multiple servers and regions for application resilience, scalability, and intelligent geo-placement.

Meeting these challenges has driven developer adoption of non-tabular, sometimes called "NoSQL" databases over the past decade. There are many flavors of non-relational databases defined by the data model they support – key-value, wide-column, and graph. But it is document databases, based on JSON-like documents, that have risen to become the most popular and widely used alternative to traditional relational systems. This is because the document data model is:

- **Intuitive:** mapping to the way developers think and code.
- **Flexible:** dynamically adapting to change.
- **Universal:** allowing data to be modeled in any shape or structure (sometimes called "multi-model").
- **Powerful:** enabling developers to work with data for almost any class of application using a consistent and expressive API.

In this guide, we explore why the document data model has become so widely used by engineering teams building modern applications. We also discuss those features that differentiate document databases from one another.

Intuitive Data Model: Faster and Easier for Developers

The tabular row-and-column data model used by relational databases bears little resemblance to how data is represented in application code. In modern programming languages, the entity you want to store in the database (i.e., customer, product, order, trade, log message, sensor reading) is represented as a complete object, with all related attributes contained in a single data structure.

Typically developers must coordinate with database administrators (DBAs) to translate that rich structure in the application to one that fits the rules of the relational model. Through normalization the object's data is decomposed across many separate parent-child tables connected by foreign keys. The example in Figure 1 shows how even a lightweight customer record is split across seven separate tables.

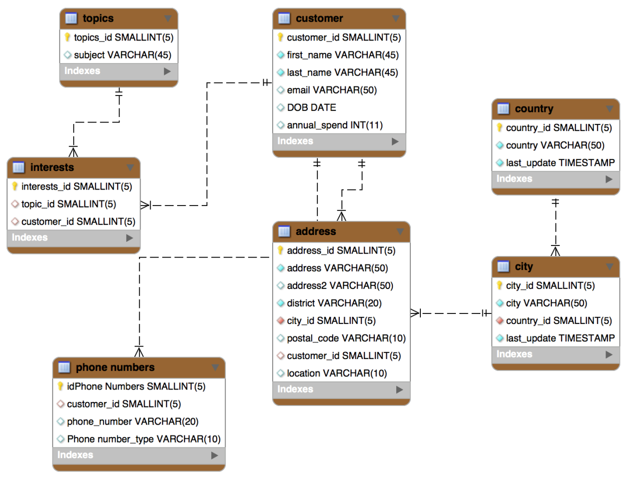

Figure 1: Customer data modeled across separate tables in a relational database

Simple applications require tens of tables, escalating to hundreds or even thousands as the application and its data becomes more complex.

This difference in data structures, sometimes referred to as "object-relational impedance mismatch", makes it difficult for developers to reason about the underlying data model while writing code, adding friction to application development.

One workaround is to introduce an Object-Relational Mapping (ORM) layer into the development stack. But this creates its own challenges, including managing the middleware and revising the mapping whenever either the application code or the database schema changes. These changes typically need to be coordinated across multiple engineering teams – developer, DBA, and Ops – creating complex dependencies that are at odds with modern development practices.

A further challenge is that ORMs do not always produce the most efficient or performant queries, and the abstraction they present makes it difficult for developers to optimize the query.

How the Document Data Model is Different

In contrast to the tabular model, the document data model presents a much more intuitive and natural way to describe data. This is because documents are consistent with the way developers think and code.

Rather than splitting data apart and flattening it out across multiple tables, documents are closely aligned to the structure of objects in the programming language. Documents are single and contained data structures with related data embedded as subdocuments and arrays. In more advanced document databases like MongoDB, each element can be individually indexed and updated, no matter how deeply nested it is within the document.

The JSON document example below contrasts how our customer object modeled across separate parent-child tables in a relational database is modeled in a single, rich document structure in a document database.

```
{
    "_id": "5ad88534e3632e1a35a58d00",
    "customerID": 12345,
    "name": {
        "first": "John",
        "last": "Doe"
    },
    "address": [
        {
            "location": "work",
            "address": {
                "street": "16 Hatfields",
                "city": "London",
                "postal_code": "SE1 8DJ"
            },
            "country": "United Kingdom",
            "geo": {
                "type": "Point",
                "coord": [ 51.5065752,-0.109081 ]
            }
        }
    ],
    "email": "john.doe@acme.com",
    "phone": [
        {
            "location": "work",
            "number": "+44-1234567890"
        }
    ],
    "dob": "1977-04-01T05:00:00Z",
    "interests": [ "devops", "data science" ],
    "annualSpend": 1292815.75
}
```

As a result of the document model approach it's simpler and faster for developers to model how data in the application will map to data stored in the database. It also significantly reduces the barrier-to-entry for new developers who begin working on a project – for example, adding new microservices to an existing application.

Beyond Developer Speed: Application Speed

Normalizing data in the tabular model means that accessing data for an entity, such as our earlier customer record example, requires JOINing multiple tables together. JOINs incur a performance penalty, even when optimized – which takes time, effort, and advanced SQL skills.

Documents, on the other hand, present a single place for the database to read and write data for an entity – what is stored together is typically accessed together. This locality of data ensures the complete document can be returned in a single database operation, avoiding the need internally to retrieve data from many different tables and rows.

Should your application access patterns require it, some document databases like MongoDB provide the ability to JOIN data between multiple collections (analogous to tables in a relational database), and to UNION complete collections. This provides additional flexibility for analytics workloads, but is generally not required for most transactional use cases.

Data Typing

Many relational databases that have been retrofitted with support for JSON columns simply store data as primitive JSON data types made up of strings and numbers, or even worse, as opaque BLOBs. There are many native document databases that do the same.

MongoDB on the other hand extends the JSON representation with the BSON (Binary JSON) encoding to include additional data types such as int, long, floating point, data, and decimal128.

This makes it much easier for developers to reliably index, process, sort, and compare more advanced data types representing things like monetary values, geospatial coordinates, and time-series data. This approach also improves the portability of documents between different programming languages.

Flexible Schema: Dynamically Adapt to Change

The relational data model is built for tabular data where each record in a table has identical columns. While it's possible to handle polymorphism and semi-structured data, it's inefficient as every row must include columns that may rarely be populated with data. Working around the basic structural limitations of flat tables consumes development time, requiring lots of additional code.

In the relational model, the schema for each table must also be known in advance and predefined before any data can be inserted into the database. Practically this means that developers and DBAs need to define their data model early in the development cycle. Any subsequent changes to the data model then requires complex schema migrations that need to be coordinated across multiple teams. Schema changes can add performance overhead during the migration process, and in some cases, can even take the database offline.

92% of respondents in a survey of DevOps professionals reported that it was difficult to accelerate the deployment of database schema changes in an effort to match the pace at which they deploy application code changes. The survey showed that just under 60% of all application changes require modifications to an existing schema, and that database changes take longer to deploy than application changes. Consider also that in the same survey:

- 43% of respondents reported they are releasing changes daily or weekly.
- Respondents lost hours or days reviewing database change scripts.
- Even after these reviews, 84% had serious production issues due to database change errors.
- 88% took more than an hour to resolve these issues.

Even trivial modifications to an existing relational schema results in a complex dependency chain – from updating ORM class-table mappings to programming language classes that have to be recompiled and application code changed accordingly. All of these steps have to be coordinated across different engineering teams, consuming both developer and DBA time and adding cycles and cost to the release process.

To avoid this complexity, some developers may be tempted, or even be recommended to overload the meaning of existing schema attributes rather than adding new ones. This runs the risk of obfuscating the code and building technical debt.

What these issues show is that the relational database's rigid, tabular data model is a poor match for today's agile, iterative development processes and continuous delivery pipelines.

How the Document Data Model is Different

The document data model offers a number of properties that make it flexible and dynamic, but which also enable you to maintain similar levels of schema control that relational databases have traditionally afforded.

Firstly the document model is polymorphic – fields can vary from document to document within a single collection. For example, you can enforce that all customer documents contain the customer ID, name, address, and the date they opened their account. But you may have additional attributes for only some of your customers such as their social media handle, interests, or location data from a mobile app. Documents make modeling such diverse attributes easy for developers, elegantly handling data of any structure.

Secondly, there is no need to declare the structure of documents to the database – documents are self-describing. Developers can start writing code and persist objects as they are created.

Thirdly, when you need to make changes to the data model, the document database continues to store the updated objects without the need to perform costly `ALTER TABLE` operations, update a separate ORM middleware layer, and coordinate all of these changes across multiple developer, DBA, and Ops teams. Documents allow multiple versions of the same schema to exist in the same table space. Old and new applications can co-exist.

Through these advantages, the flexibility of the document data model is well suited to the demands of modern application development practices.

Schema Governance

While a flexible schema is a powerful feature, there may come a time in an application's lifecycle – for example when it's functionality has reached steady state – that you want more centralized control over the data structure and content of your database.

Most document databases push enforcement of these controls back to the developer to implement in application code. However more advanced document databases provide schema validation, using approaches such as the IETF JSON Schema standard adopted by MongoDB.

Using MongoDB's schema validation, you can define a prescribed document structure for each collection, with the database configured to either reject or log any documents that do not conform to it. You can enforce the presence of mandatory fields, define data types and permissible field values, and optionally block the addition of new fields that have not been explicitly approved by the application owner.

With schema validation, you have control to apply data governance standards to a document schema when an application is in production, while maintaining the benefits of a flexible data model in development.

Universal: JSON Documents for Multi-Model

Lightweight, language-independent, and human readable, JSON has become an established standard for data interchange and storage. JSON is used across the application stack – from frontend web and mobile services, to inter-service messaging, streaming and APIs, through to the backend server and data storage. You don't have to transpose, serialize and deserialize your data between different formats in each layer of your application stack, making development much more fluid.

Superset of all Data Models

Wherever you use them in your application, a major advantage of JSON documents is that you have the flexibility to model your data any way your application needs.

The nesting of arrays and subdocuments makes documents very powerful at modeling complex relationships between data. But you can also model flat, tabular and columnar structures, simple key-value pairs, text, geospatial and time-series data, or the nodes and edges of a connected graph data structure. Because of this flexibility, documents are a superset of all these different data models.

Rather than use multiple databases each supporting a specific data model for each part of your application, the multi-model approach offered by document databases significantly enhances developer productivity, eliminates data duplication, and reduces operational costs:

- Developers work with a single query language across all data models, providing a consistent development experience.
- Rather than trying to run multiple databases, you operationalize one platform with consistent controls for resilience, scalability, and security.

Review our Building with Patterns blog series to learn more about schema design best practices for different categories of use-case.

Powerful: Query Data Any Way You Need. ACID without Compromise

A major point of differentiation between document databases is how they allow you to query and manipulate data. Some offer little more than key-value and range queries. They also require you to rewrite a complete copy of the document, even if you are just modifying a single element in an array.

More advanced document databases offer additional capabilities that allow you to filter, sort, aggregate and update any fields, no matter how deeply nested they are within a document.

The MongoDB Query Language (MQL) and aggregation pipeline is comprehensive and expressive. You can compose powerful queries and federate them across all your data wherever it lives – in databases and search for transactional applications, and in your data lake for operational analytics.

As illustrated in Table 1, MQL gives you much more than just simple lookups. You can create sophisticated processing pipelines for data analytics and transformations with aggregations, JOINs, UNIONs, geo, text, and graph search, and on-demand materialized views.

Expressive Queries	▪ Find anyone with phone # "1-212…" ▪ Check if the person with number "555…" is on the "do not call" list
Geospatial	▪ Find the best offer for the customer at geo coordinates of 42nd St. and 6th Ave
Text Search	▪ Find all tweets that mention the firm within the last 2 days
Faceted Navigation	▪ Filter results to show only products <$50, size large, and manufactured by ExampleCo
Aggregation	▪ Count and sort number of customers by city, compute min, max, and average spend
Native Binary JSON Support	▪ Add an additional phone number to Mark Smith's record without rewriting the document at the client ▪ Update just 2 phone numbers out of 10 ▪ Sort on the modified date
Fine-grained Array Operations	▪ In Mark Smith's array of test scores, update every score <70 to be 0
JOIN ($lookup)	▪ Query for all San Francisco residences, lookup their transactions, and sum the amount by person
Graph Queries ($graphLookup)	▪ Query for all people within 3 degrees of separation from Mark

Table 1: MongoDB's rich query functionality

The SQL to MongoDB mapping chart is a useful resource for anyone coming to MongoDB from a traditional SQL development background. It provides examples of SELECT, INSERT, UPDATE, DELETE and table level SQL statements and contrasts them directly to the equivalent MongoDB Query Language syntax.

The SQL to Aggregation mapping chart provides an overview of common SQL aggregation terms, functions, and concepts and the corresponding MongoDB aggregation operators. MongoDB's aggregation framework is modeled on the concept of data processing pipelines. Documents enter a multi-stage pipeline that transforms the documents into an aggregated result. If you have a query that returns a set of documents and you then want to transform them, you simply add new stages as needed. In the case of SQL, you'd need to rewrite the entire query from scratch.

For the purposes of illustration, consider the final example in the SQL to Aggregation mapping chart where you are looking to count the number of distinct customer IDs and order_date groupings.

Using SQL, you would first have to write a subquery to count the distinct groupings. If you then wanted to group the output into buckets – for example counting all customers who place one order per day, counting those who place two to five orders per day, etc – it becomes highly complex in SQL. Using the MongoDB aggregation pipeline, you simply add a new stage.

Indexing Documents

For your queries to efficiently access data, you should be able to declare indexes on any field within your documents, including fields nested within arrays.

Some document databases support only indexes against a document's primary key, limiting how you can access data without resorting to slow full table scans. Other document databases support only a limited range of secondary indexes that are maintained asynchronously from base document data, introducing the complexity of eventual consistency to your applications.

MongoDB overcomes these limitations by offering a broad range of index types and features with language-specific sort orders to support complex access patterns to your data. MongoDB indexes can be created and dropped on-demand to accommodate evolving application requirements and query patterns.

Index Types	Index Features
Primary Index: Every Collection has a primary key index	**TTL Indexes**: Single Field indexes, when expired delete the document
Compound Index: Index against multiple keys in the document	**Unique Index**: Ensures value is not duplicated
MultiKey Index: Index into arrays	**Partial Index**: Expression based indexes, allowing indexes on subsets of data
Lucene Index: Support for full-text search (MongoDB Atlas)	**Case Insensitive Indexes**: supports text search using case insensitive search
GeoSpatial Indexes: 2d & 2dSphere indexes for spatial geometries	**Sparse Index**: Only index documents which have the given field
Wildcard Index: Auto-index all matching fields & nested elements	

Table 2: MongoDB offers fully-featured secondary indexes

Working with Document Data

To accelerate developer productivity, MongoDB provides native drivers for all popular programming languages and frameworks.

Supported drivers include Java, Javascript, C#/.NET, Go, Python, PHP, Rust, Scala and others. All supported MongoDB drivers are designed to be idiomatic for the given programming language – with syntax and behaviors that are familiar to developers using those languages. This makes it much more natural for you to work with the database than string-based languages like SQL.

The documentation includes examples of MongoDB operations in supported languages – for example inserting one or multiple documents into a collection. In addition, our Developer Quick Start channel provides deeper tutorials in using the MongoDB Query Language with our drivers across all of the leading programming languages.

Beyond programmatic access, you can also interact with MongoDB graphically using MongoDB Compass, the free GUI for MongoDB. Through Compass you can explore your schema with histograms that show your documents' fields, data types, and values. You can visually create queries and aggregation pipelines from the GUI and then export them as code to your app; manipulate data, view and create indexes; build schema validation rules and views; and more.

Much of this functionality is also available to you directly within your favorite IDEs, including VS Code. These integrations enable MongoDB to fit seamlessly into your native workflow and development tools.

Tools and Connectors

In the digital economy, it is vital that developers, business analysts, and data scientists can extract insights from data – whether that is for traditional reporting and BI or to build more intelligent applications with machine learning. MongoDB provides a range of visualization tools and connectors to make this straightforward:

- MongoDB Charts is the fastest and easiest way to create visualizations of MongoDB data. You can construct graphs and build dashboards, sharing them with other users for collaboration, and embed them directly into your web apps to create engaging, data-driven user experiences.
- The MongoDB Connector for BI lets you use MongoDB as a data source for your existing SQL-based BI and analytics platforms such as Tableau, Microstrategy, Looker, Excel, and more, without having to perform any ETL operations.
- The MongoDB Connector for Apache Spark exposes MongoDB data to all of Spark's libraries, including

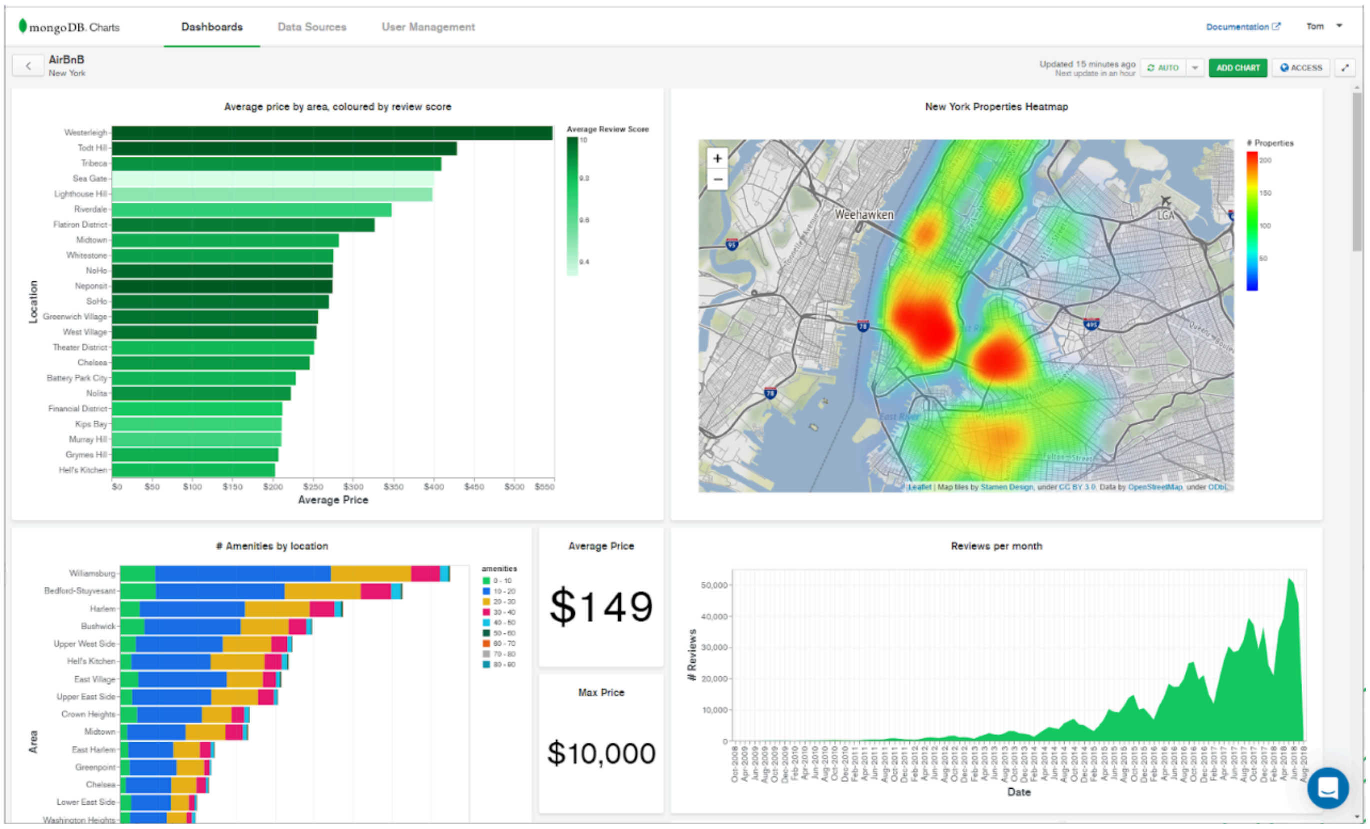

Figure 2: Creating rich visualizations of your data with MongoDB Charts

Scala, Java, Python and R. MongoDB data is materialized as DataFrames and Datasets for integration with machine learning frameworks.

To make it easy for businesses to act on data in real time, many developers are building fully reactive, event-driven data pipelines. MongoDB goes beyond many other databases with features like Change Streams that allow applications to access real-time data changes in the database, while MongoDB Atlas Triggers allow you to execute server-side logic in response to database change events. This might be updating related data in other documents, or calling functions that, for example, send an email to a new customer when they sign up to your service, or firing an alert to a user's mobile app when a large charge is made to their account.

With the MongoDB Connector for Apache Kafka, you can build robust data pipelines that move events between systems in real time, using MongoDB as both a source and sink for Kafka. The connector is supported by MongoDB and verified by Confluent.

Data Consistency and Transactional Integrity

While documents make it much easier and faster to model an application's data, developer productivity can be compromised if they then have to struggle to enforce data consistency. Some document databases are designed on a concept of "eventual consistency", where there are no guarantees that the application will read the latest version of the data written to the database. This forces you to write complex correctness code if you want to ensure data quality.

More advanced document databases like MongoDB enforce strong consistency, applying consistency guarantees to both base data, as well as to native secondary indexes. Strong consistency ensures the application can not read data that is potentially stale or that was previously deleted by another client.

ACID Transactions

Most document databases offer atomic guarantees for writes to a single document. This is suitable for many applications because the document model brings together related data that would otherwise be modeled across separate parent-child tables in a tabular schema.

With single document atomicity, one or more fields may be written in a single operation, including updates to multiple subdocuments and elements of an array. These guarantees ensure complete isolation as the document is updated; any errors cause the operation to roll back so that clients receive a consistent view of the document.

However not every application can be served with atomicity that is scoped to only a single document. This is why some document databases go further by offering support for multi-document ACID transactions. Many implement multi-document transactions via server-side stored procedures or in client-side code. In both cases, the developer still has to write the logic to control the consistency and isolation necessary to enforce ACID properties.

MongoDB takes a different approach by implementing multi-document transactions in a way that is consistent with relational databases, making them familiar to developers coming from a SQL background.

- They are fully conversational from the application (whereas some databases insist that all reads are made before the first write).
- They also present a similar syntax, supporting multiple statements with all-or-nothing execution and snapshot isolation. To ensure these guarantees, it is important developers apply the appropriate read and write concerns for their transactions.

To show how straightforward it is to use transactions in MongoDB, the following code snippets compare the transactions syntax for a relational database and the MongoDB Query Language

SQL

```
START TRANSACTION;
INSERT INTO orders (...) VALUES (...);
UPDATE stock SET quantity=... WHERE ...;
COMMIT;
```

MongoDB Query Language (MQL)

```
session.startTransaction();
db.orders.insert ({....})
db.stock.update ({ ... } },
    { $set: { quantity: ... } }})
session.commitTransaction();
```

The availability of multi-document ACID transactions makes it even easier for developers to address a complete range of use cases with MongoDB.

Foreign Keys and Referential Integrity

Foreign keys are necessary to maintain referential integrity in tabular schema models that split data and its relationships up across multiple tables.

With documents, referential integrity is in-built to the rich, hierarchical structure of the data model. When modeling a parent-child or 1:many relationship with subdocuments or arrays, there is no way you can have an orphan record – related data is embedded inside a document so you know the parent exists.

Another use of foreign keys is to verify that the value of a specific field conforms to a range of permissible values – e.g., country names or user status. You can do this with MongoDB's schema validation as data is written to the database, avoiding the need to re-verify the data whenever you retrieve it.

Distributed: Resilient and Globally Scalable

Unlike monolithic, scale-up relational databases, most document databases are distributed systems by design.

As illustrated earlier, rather than the flat tabular data model with masses of inter-relationships between tables and rows, documents are single, self contained data structures. Therefore it is much easier to distribute documents across multiple servers while preserving the locality of related data in a single, rich document structure and maintaining performance as the application scales.

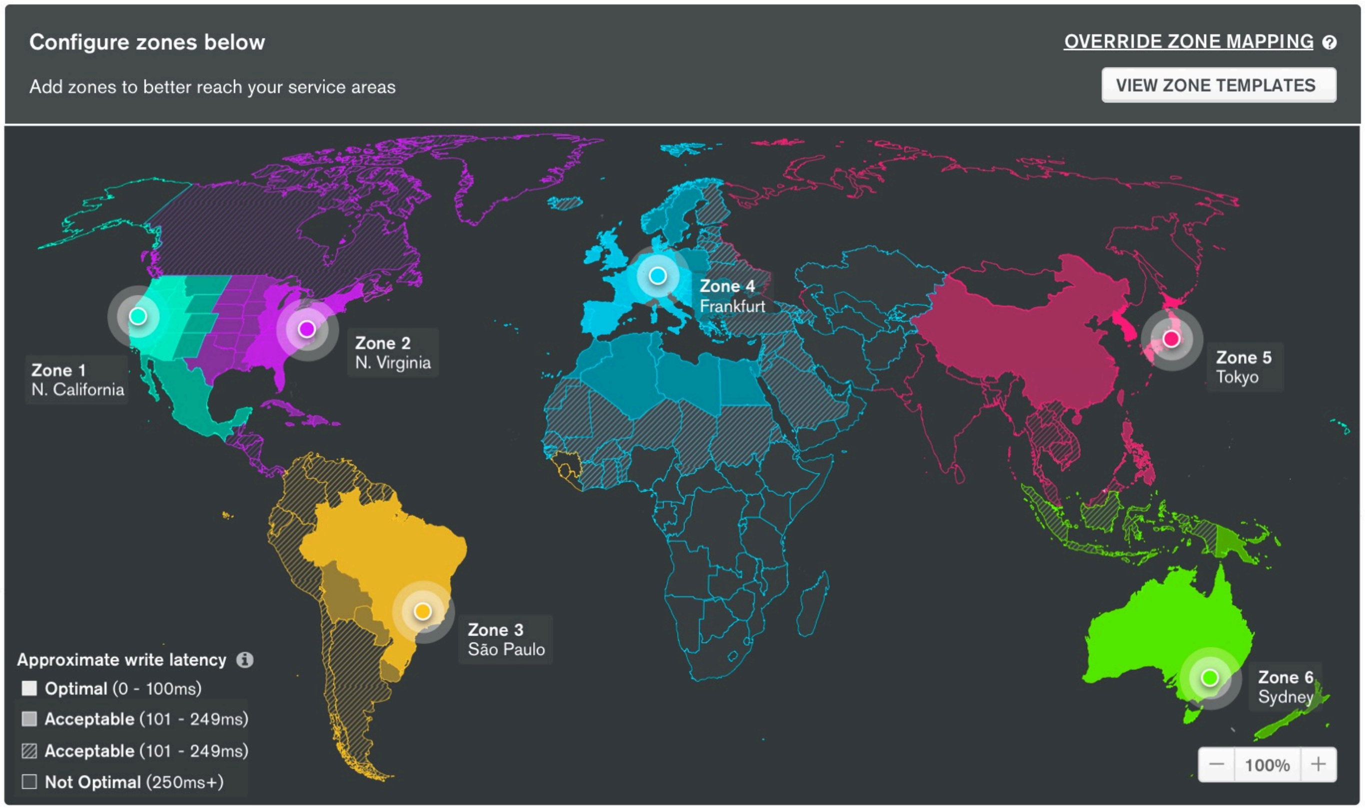

Figure 3: Serving always-on, globally distributed, write-everywhere apps with MongoDB Atlas Global Clusters

Through replication and sharding, high availability, horizontal scaling, and geographic distribution are all built into the database and easy to use.

Replication: No Single Point of Failure

MongoDB replica sets enable you to create up to 50 copies of your data, which can be provisioned across separate servers and geographic regions. With self-healing recovery, you have resilience to outages and planned maintenance events. Replica sets also enable you to:

- Scale read operations, intelligently routing queries to a copy of the data that is physically closest to the user.
- Isolate different workloads on a single cluster. With MongoDB Atlas Analytics Nodes you can share the same data in real time across transactional and analytical applications, isolating the workloads so they never contend for resource with one another.

Sharding for Horizontal Scale Out

Through native sharding, MongoDB can scale your database out across multiple nodes to handle write-intensive workloads and growing data sizes. Sharding with MongoDB allows you to seamlessly scale the database as your applications grow beyond the hardware limits of a single server, and it does so without adding complexity to the application.

To respond to evolving workload demands, you can add and remove shards at any time. You also have the flexibility to refine your shard key on demand, without impacting system availability. As your shard key is modified or as you change the cluster topology, MongoDB will automatically rebalance data across shards as needed without manual intervention.

By simply hashing a primary key value, most document databases randomly spray data across a cluster of nodes. This imposes performance penalties when data is queried, and adds application complexity when data needs to be localized to a specific region.

By exposing multiple sharding policies, MongoDB offers you a better approach. With ranged, hashed, and zoned sharding, you can partition your data based on query patterns or data placement requirements, giving you much higher scalability across a diverse set of workloads.

Global Clusters in MongoDB Atlas – the fully managed cloud database service – allows you to quickly implement zoned sharding using a visual UI or the Atlas API. You can easily create distributed databases to support geographically distributed apps, with policies enforcing data residence within specific regions.

Tiered Scaling

Beyond horizontal scaling, MongoDB also offers tiered scaling. When working in the cloud, MongoDB Atlas Online Archive will automatically tier aged data out of the database onto cloud object storage in the Atlas Data Lake. Archived data remains fully accessible with federated queries that span both object and database storage in a single connection string.

This approach enables you to more economically scale data storage by moving it to a lower cost storage tier without losing access to the data, and without grappling with slow and complex ETL pipelines.

JSON in Relational Databases

With document databases empowering developers to move faster, most relational databases have added support for JSON columns. However, simply adding a JSON data type does not bring the benefits of a native document database. This is because the relational approach detracts from developer productivity, rather than improving it. These are some of the things you will have to deal with:

- **Proprietary Extensions**: Working with documents means using custom, vendor-specific SQL functions which will not be familiar to most developers, and which don't work with the broad ecosystem of SQL tools. Add low-level JDBC/ODBC drivers and ORMs and you face complex development processes and low productivity.
- **Primitive Data Handling**: Presenting JSON data as simple strings and numbers rather than the rich data types supported by native document databases such as MongoDB makes computing, comparing, and sorting data complex and error prone.
- **Poor Data Quality & Rigid Tables**: Relational databases offer little to validate the schema of documents, so you have no way to apply quality controls against your JSON data. And you still need to define a schema for your regular tabular data, with all the overhead that comes when you need to alter your tables as your application features evolve.
- **Low Performance**: Most relational databases do not maintain statistics on JSON data, preventing the query planner from optimizing queries against documents, and you from tuning your queries.
- **No scale-out**: Traditional relational databases offer no way for you to partition ("shard") the database across multiple instances to scale as workloads grow. Instead you have to implement sharding yourself in the application layer, or rely on expensive scale-up systems.

Use Cases for Document Databases

With the ability to work natively with JSON, web and mobile backends were some of the first use cases for document databases. The flexibility of the data model and distributed systems design also appealed to developers working on fast evolving applications – domains such as content management, product catalogs, user profiles, and logging, represented the next wave of use cases.

All of these remain important applications. But as document databases have rapidly matured – adding more powerful query engines, offering stronger durability and transactional guarantees, hardened security controls, and more – so document databases are now being used for an even broader range of applications.

Common use cases for MongoDB include the following:

- Customer data management and personalization: Expedia, YouGov
- Single view (of customer, of city): Barclays, City of Chicago

- IoT and time-series data: Bosch, Mercedes-Benz
- Trading and payments: HSBC, Coinbase
- Ecommerce: Cisco, OTTO
- Product catalogs and content management: eBay, Adobe
- Gaming: Sega, EA
- Mobile Apps: 7-Eleven, ADP
- Mainframe Offload: Alight Solutions, formerly Aon Hewitt, LCL
- Online Analytics and AI: KPMG, Continental AG

You can learn more about use cases that are a good fit for MongoDB, and those where you should evaluate alternative solutions from our Use Case Guide.

Summary

The use of document databases such as MongoDB has exploded over the past decade, supporting a range of transactional, operational, and analytical workloads.

The document data model is intuitive, flexible, universal, and powerful, enabling developers to build applications faster, and scale them further than traditional relational databases. This is why MongoDB has been rated as the database developers most want to work with in the Stack Overflow Developer Survey for the past four years.

As most technologies in use today have moved on from the designs of the 1970s and 80s, it's about time your databases did as well. This is why documents represent the largest advance in database design in a generation.

Getting Started

The best way to explore the power of the document model is to spin up MongoDB on the fully-managed Atlas cloud service. Our documentation steps you through how to create a free MongoDB database cluster in the region and on the cloud provider of your choice. It includes steps on how to load our pre-prepared sample datasets, providing you with a simple way of getting started with documents.

You should also register for the MongoDB University which provides no-cost, web-based training on the basics of MongoDB, and provides a curated learning path for developers covering data modeling through to building your first application using the most popular programming languages.

We Can Help

We are the company that builds and runs MongoDB. Over 18,400 organizations rely on our commercial products. We offer cloud services and software to make your life easier:

MongoDB Atlas is the global cloud database service for modern applications. Deploy fully managed MongoDB across AWS, Azure, or Google Cloud with best-in-class automation and proven practices that guarantee availability, scalability, and compliance with security standards.

MongoDB Enterprise Advanced is the best way to run MongoDB on your own infrastructure. It's a finely-tuned package of advanced software, support, certifications, and other services designed for the way you do business.

MongoDB Atlas Data Lake allows you to quickly and easily query data in any format on Amazon S3 using the MongoDB Query Language and tools. You don't have to move data anywhere, you can work with complex data immediately in its native form, and with its fully-managed, serverless architecture, you control costs and remove the operational burden.

MongoDB Charts is the best way to create, share and embed visualizations of MongoDB data. Build visualizations quickly and easily to analyze complex, nested data. Embed individual charts into any web application or assemble them into live dashboards for sharing.

Realm Mobile Database allows developers to store data locally on iOS and Android devices using a rich data model that's intuitive to them. Combined with the MongoDB Realm sync-to-Atlas, Realm makes it simple to build reactive, reliable apps that work even when users are offline.

MongoDB Realm allows developers to validate and build key features quickly. Application development services like Realm Sync for mobile and Realm's GraphQL service, can be used with Realm Functions, Triggers, and Data Access Rules – simplifying the code required to build secure and performant apps.

MongoDB Cloud Manager is a cloud-based tool that helps you manage MongoDB on your own infrastructure. With automated provisioning, fine-grained monitoring, and continuous backups, you get a full management suite that reduces operational overhead, while maintaining full control over your databases.

MongoDB Consulting packages get you to production faster, help you tune performance in production, help you scale, and free you up to focus on your next release.

MongoDB Training helps you become a MongoDB expert, from design to operating mission-critical systems at scale. Whether you're a developer, DBA, or architect, we can make you better at MongoDB.

Resources

For more information, please visit mongodb.com or contact us at sales@mongodb.com.

Case Studies (mongodb.com/customers)
Presentations (mongodb.com/presentations)
Free Online Training (university.mongodb.com)
Webinars and Events (mongodb.com/events)
Documentation (docs.mongodb.com)
MongoDB Atlas database as a service for MongoDB (mongodb.com/cloud)
MongoDB Enterprise Download (mongodb.com/download)
MongoDB Realm (mongodb.com/realm)

mongoDB

US 866-237-8815 • INTL +1-650-440-4474 • info@mongodb.com