2025

A Technical Guide for RDBMS to MongoDB Migration

Discover our proven application database migration process and tools to successfully migrate to MongoDB

Table of Contents

What Is in This Guide?

While traditional relational databases have served enterprises for decades, their rigid schemas, scaling limitations, and inability to handle diverse data types make them an inflexible solution for modern applications. These limitations present significant challenges for technology teams tasked with delivering on today's business goals.

MongoDB experts developed this guide to help application modernization teams, cloud architects, and database engineers effectively use the document model to meet modern application needs.

The guide details a step-by-step migration process centered on the MongoDB Relational Migrator—a free developer tool that uses generative AI to automate the most complex aspects of migration, which significantly reduces effort and minimizes risk.

Finally, for teams seeking additional assistance, this guide introduces the dedicated professional services and training programs that can support your data migration journey.

Introduction

The relational database has been the foundation of enterprise data management for more than 50 years. However, the way we build and run applications today, coupled with unrelenting growth in new data sources and user loads, is pushing relational databases beyond their limits. This can inhibit business agility, limit scalability, and strain budgets, compelling more and more organizations to migrate to alternatives.

MongoDB is designed to meet the demands of modern applications with a technology foundation that enables you through:

Flexible architecture, suitable for AI applications: MongoDB's document model offers a flexible way to represent complex, hierarchical data structures, while ensuring reliable and consistent database transactions, unlike the rigid format of traditional relational databases.

Faster innovation: Instead of scattering data across multiple tables in a relational database, MongoDB allows you to store data in a single document. This powerful simplicity gives organizations the flexibility to easily adapt data models without application downtime, enabling them to innovate quickly as new application requirements and use cases emerge.

Freedom to run anywhere: MongoDB Atlas is a fully managed cloud database that is available in more than 125 regions across major cloud providers. Organizations are not locked into a single cloud provider and enjoy greater flexibility for data residency and compliance requirements.

Integrated features: MongoDB Atlas unifies powerful features into one platform, including vector and full-text lexical search (Atlas Vector Search/Atlas Search), data visualization (Atlas Charts), data stream processing (Atlas Stream Processing) for event-driven applications, and more.

Best-in-class retrieval to build trustworthy AI applications: Leverage Atlas Vector Search with advanced embedding and reranking models from Voyage AI (now part of MongoDB) for semantic search and retrieval-augmented generation (RAG). This enables AI applications to access current, relevant data and simplifies workflows for developing intelligent agents and AI-powered solutions—all within a unified platform. Use MongoDB's Model Context Protocol (MCP) Server to connect AI agents and assistants directly to your MongoDB instance.

Security and compliance: MongoDB Atlas offers innovations like full-lifecycle data encryption: in transit, at rest, and—with our industry-leading Queryable Encryption technology—even while in use.

This guide is designed for project teams that want to know how to migrate from a relational database management system (RDBMS) to MongoDB.

This document also provides various links to help users find appropriate online resources. For the most current and detailed information on particular topics, please see MongoDB's documentation.

Choosing the Right Path

MongoDB offers various solutions to help customers transition from traditional relational databases to its platform. One option is Relational Migrator, a free self-service tool designed for customers with in-house migration expertise. For larger and more complex migrations, MongoDB's Professional Services team is available to provide assistance throughout the entire migration process.

Relational Migrator is a free developer tool that leverages intelligent algorithms and generative AI to automate the most complex aspects of migrating from traditional relational databases to MongoDB. It significantly reduces the time and effort required for migration while minimizing associated risks.

If you have large-scale, complex migrations for business-critical applications, our Professional Services team can provide dedicated, end-to-end migration support. Our experts work closely with you to accelerate and de-risk your migration, providing strategic and technical guidance and development resources to execute your migration initiatives.

Multiple organizations from various industries have successfully migrated from RDBMS to MongoDB for a range of applications, including:

Organization	Industry	Migrated From (RDBMS)	Application Migrated
Bendigo and Adelaide Bank	Financial Services	SQL Database	**Application:** Agent Delivery System **Function:** A retail teller application for customer service transactions
Nationwide Building Society	Financial Services	SQL Server	**Application:** Overdraft database **Function:** Manages customer overdraft information and transactions
Powerledger	Renewable Energy	SQL Database	**Application:** Energy trading data platform **Function:** Blockchain-based renewable energy trading platform
Toyota Connected Services	Automotive	SQL Database	**Application:** Toyota Safety Connect **Function:** Manages safety and emergency services for connected vehicles
Lombard Odier	Financial Services	SQL Database	**Application:** Portfolio management system **Function:** Manages client assets and investment portfolios for the bank
SEGA	Gaming	MySQL	**Application:** Customer portals **Function:** Enhances gamer engagement with promotions, forums, and stats
OXY	Energy	Microsoft SQL Server	**Application:** Automated land-lease agreement management solution **Function:** Automates classification and management of land-lease documents
TELUS Health	Healthcare Services	SQL Database	**Application:** TELUS Health One **Function:** Provides holistic well-being support and Employee Assistance Program services
Cathay Pacific	Transportation and Logistics	Manual Documents	**Application:** Flight Folder **Function:** Digitized flight operations for the crew, replacing paper documents

Figure 1: Case studies.

See more examples of how MongoDB is helping enterprises modernize legacy applications here.

Application Database Migration Process With MongoDB

MongoDB's application database migration process follows a structured, phased approach to migrating applications running on relational databases to its document database. The six-phase framework—assess, understand, design, implement, validate, and optimize—developed by MongoDB Professional Services enables teams to prioritize high-impact applications that are ready for database migration, redesign schemas for flexibility, validate performance, and optimize incrementally. This method balances risk management with agile execution, using tools like MongoDB Relational Migrator to streamline migrations with minimal downtime. This process ensures that customers can access the benefits of a modern database like MongoDB, rather than simply replicating their relational data patterns in a document database.

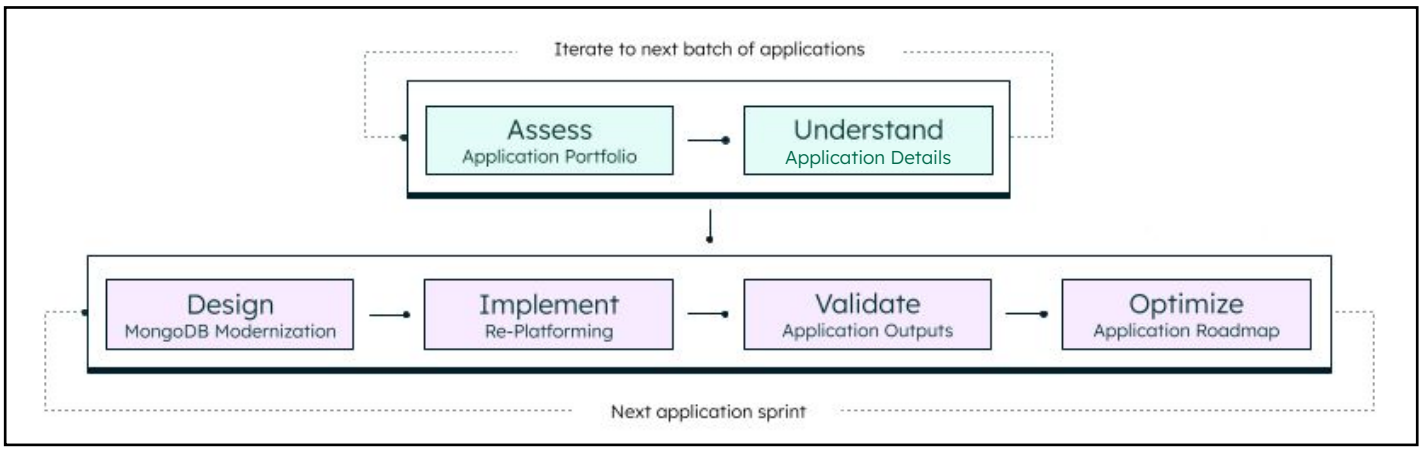


Figure 2: Application database migration process with MongoDB.

Here are details of each step of the process:

- **Assess application portfolio**
  Teams begin by assessing their current applications in groups, scoring each based on how well they align with MongoDB's document model. This phase helps identify the highest-priority applications for migration—especially those that would gain the most from the flexibility offered by the document model, such as applications that require dynamic schemas or global availability.

- **Understand the application details**
  For shortlisted apps, teams analyze data structures, dependencies, and infrastructure. This step involves creating a high-level MongoDB schema design and making a final "go/no-go" decision based on technical feasibility and business impact.

  - **Bonus tip:** MongoDB's Relational Migrator has a pre-migration analysis feature that can automate this stage and help you save valuable time and resources. It uses advanced algorithms to analyze the source schema, highlight potential data and configuration risks, and provide tailored recommendations to help you successfully migrate to MongoDB. You can learn more about this feature on our product page.

- **Design for migration to MongoDB**
  Architects re-model relational tables into MongoDB collections using document patterns (like embedding vs. referencing) and create a detailed execution plan to address schema changes, data migration workflows, and integration points.
    - **Bonus tip:** Relational Migrator offers a user-friendly schema mapping feature for architects to visualize, compare, and model their relational schema to MongoDB in an entity-relationship diagram. You can learn more about this feature in the data modeling section of our documentation.
- **Implement the migration**
  Migration teams initiate the migration process using ETL tools or by writing custom scripts.
    - **Bonus tip:** Relational Migrator can help you easily migrate to MongoDB Atlas or Enterprise Advanced using a one-time snapshot or continuous sync with change data capture (CDC). Relational Migrator can also integrate with Apache Kafka or Confluent Cloud for large-scale extended migrations.
- **Validate application outputs**
  Teams test performance against SLAs (e.g., query latency) and verify data integrity post-migration. Production validation ensures the application behaves as expected in real-world scenarios.
    - **Bonus tip:** Relational Migrator can assist with this stage as well. You no longer need manual code updates. You can quickly generate MongoDB-compatible application code using the code generator feature. Additionally, you can use the query converter feature and leverage generative AI to convert SQL queries, views, and stored procedures to MongoDB query syntax and validate them.
- **Optimize application roadmap**
  Post-migration reviews identify optimization opportunities, such as indexing adjustments. A long-term roadmap is developed to govern future enhancements and leverage MongoDB-specific features like aggregation pipelines.

The process is iterative, using lessons from previous migrations to maintain continuity for current systems. This phased approach balances the risks of migration, allowing enterprises to transition at their own pace.

Getting Started: Concept Mapping

The most fundamental change in migrating from a relational database to MongoDB is the way in which the data is modeled.

As with any data modeling exercise, each use case will be different, but there are some general considerations that you can apply to most schema migration projects. Before exploring schema design, see Figure 3, which provides a reference comparing terminology between relational databases and MongoDB.

SQL Terms / Concepts*	MongoDB Terms / Concepts
Database	Database
Table	Collection
Row	Document
Column	Field
Index	Index
Table Joins	Embedded document, document references, or $lookup to combine data from different collections
Primary Key: Specify any unique column or column combination as the primary key	Primary Key: In MongoDB, the primary key is automatically set to the _id field
Aggregation (e.g., GROUP BY)	Aggregation pipeline
SELECT INTO NEW_TABLE	$out

***For other SQL commands, see the SQL to Aggregation Mapping Chart**

Figure 3: SQL-to-MongoDB document model terminology translation.

Schema design using documents involves a shift in perspective for data architects, developers, and DBAs. Instead of relying on the traditional relational data model, which flattens data into rigid two-dimensional tabular structures made up of rows and columns, MongoDB provides a more dynamic document data model. This model allows for rich, hierarchical data structures, including embedded sub-documents and arrays. This data modeling article highlights common patterns for MongoDB schema design.

Flexible Architecture: From Rigid Tables to Flexible and Dynamic Documents

Much of the data we use today has complex structures that can be modeled and represented more efficiently using JSON (JavaScript Object Notation) documents rather than tables.

MongoDB stores JSON documents in a binary representation called BSON (Binary JSON). BSON encoding extends the popular JSON representation to include additional data types such as integer, decimal, long integer, and floating point.

With sub-documents and arrays, documents also align with the structure of objects at the application level. This makes it easy for developers to map data used in the application to its associated document in the database.

By contrast, trying to map the data's object representation to the tabular representation of RDBMS slows down development. Adding object relational mappers (ORMs) can create additional complexity by reducing the flexibility to evolve schemas and optimize queries to meet new application requirements.

The project team should design the schema based on application requirements, leveraging the document model's flexibility to accommodate changes in data access patterns patterns as needed over time.

In schema migrations, it may be easy to mirror the relational database's flat schema to the document model. However, this approach negates the advantages enabled by the document model's rich, embedded data structures. For example, data that belongs to a parent-child relationship in two RDBMS tables (Figure 4), would commonly be an embedded document in MongoDB.

The schema mapping feature in Relational Migrator addresses this challenge with three flexible approaches:

1. **Recommended schema:** The tool uses intelligent algorithms to suggest a MongoDB schema optimized for common patterns. Users can start with this and then customize it as needed.

2. **Direct translation:** Users start with a 1:1 relational-to-document mapping as their baseline, and then refine it as needed.

3. **Full customization:** Users can build their schema from scratch using the tool's visual interface.

Pers_ID	Surname	First_Name	City
0	Miller	Paul	London
1	Ortega	Alvaro	Valencia
2	Huber	Urs	Zurich
3	Blanc	Gaston	Paris
4	Bertolini	Fabrizio	Rome

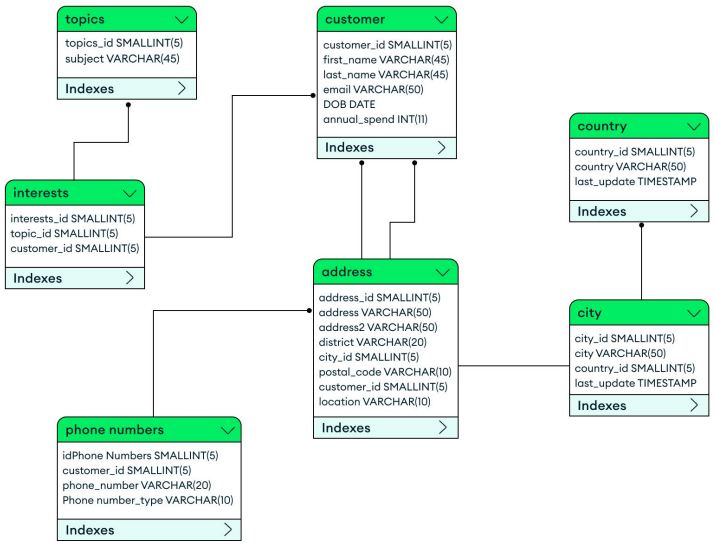
No Relation

Car_ID	Model	Year	Value	Pers_ID
101	Bentley	1973	100,000	0
102	Rolls Royce	1965	330,000	0
103	Peugeot	1993	500	3
104	Ferrari	2005	150,000	4
105	Renault	1998	2,000	3
106	Renault	2001	7,000	3
107	Smart	1999	2,000	2

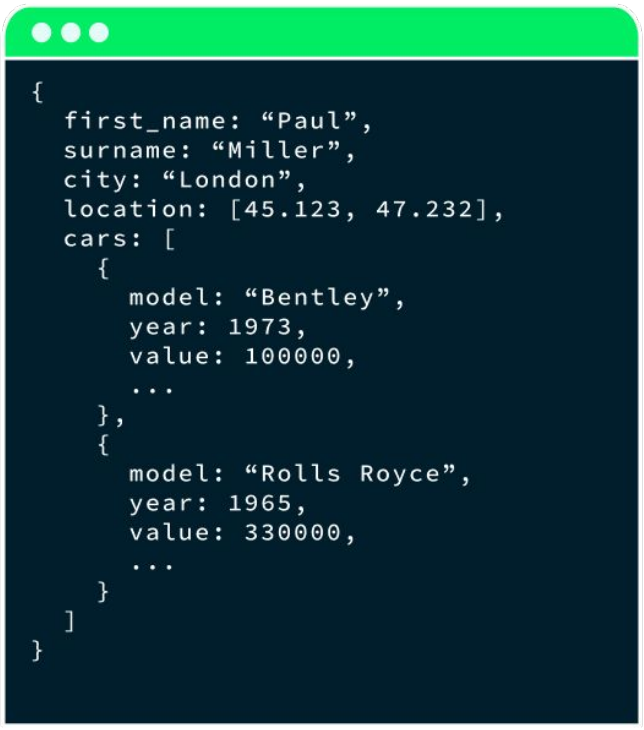
Figure 4: The RDBMS uses the Pers_ID field to join the Person table with the Car table, enabling the application to report each car's owner.

The document model effectively consolidates related fields into a single data structure using embedded sub-documents and arrays. This eliminates the need for JOIN operations, as all related information can be stored together in one document, making it easier to retrieve complete records.

To further illustrate the differences between the relational and document models, consider a slightly more complex example using a customer object, as shown in Figure 5. The customer data is normalized across multiple tables, with the application relying on the RDBMS to join seven separate tables in order to build the customer profile. With MongoDB, all of the customer data is contained within a single, rich document, collapsing the child tables into embedded sub-documents and arrays.

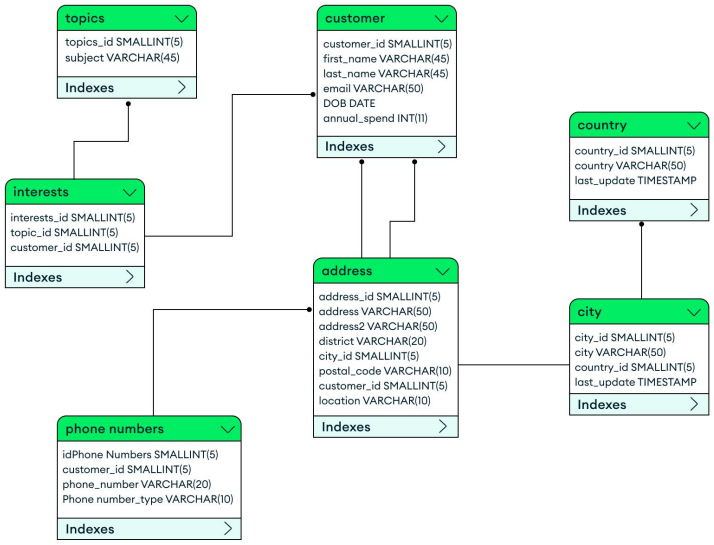


In a relational database, modeling a customer's information can be complex. The example above requires seven separate tables—including customer, address, and interests—that must be joined with common keys to create a complete customer view.

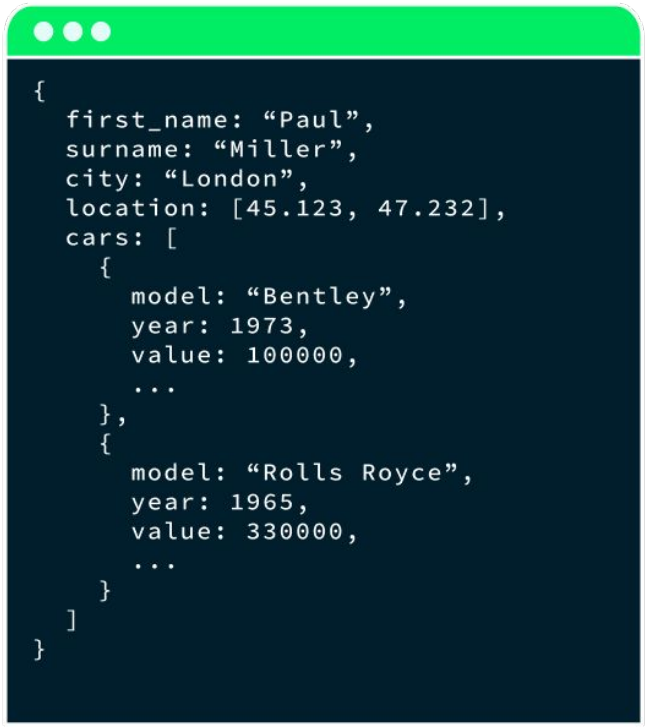


MongoDB simplifies the data model by consolidating the information from all seven tables into a single document. An array of subdocuments for each car is embedded directly within the customer document, eliminating the need for complex joins.

Figure 5: Modeling customer information: Relational database vs. MongoDB.



In a relational database, modeling a customer's information can be complex. The example above requires seven separate tables—including customer, address, and interests—that must be joined with common keys to create a complete customer view.



```
{
  first_name: “Paul”,
  surname: “Miller”,
  city: “London”,
  location: [45.123, 47.232],
  cars: [
    {
      model: “Bentley”,
      year: 1973,
      value: 100000,
      ...
    },
    {
      model: “Rolls Royce”,
      year: 1965,
      value: 330000,
      ...
    }
  ]
}
```

MongoDB simplifies the data model by consolidating the information from all seven tables into a single document. An array of subdocuments for each car is embedded directly within the customer document, eliminating the need for complex joins.

Figure 5: Modeling customer information: Relational database vs. MongoDB.

What makes the document model better for modern applications?

Data is the lifeblood of modern software applications. For over 50 years, data has been stored in RDBMS and used by developers to build software applications using structured data, organized in rigid schemas, i.e., table rows and columns.

Documents naturally reflect the structure of objects in programming languages. This simplifies data modeling for developers, allowing them to quickly map application data to the database. The result is faster data retrieval and enhanced application performance.

Modern software applications must have a few foundational features to be successful:

Scalability: Applications should be able to handle high traffic globally, large datasets, and expanded services without compromising performance or user experience.

Flexibility: Applications need to accommodate diverse data types. Developers can quickly adopt new technologies and respond to changing user demands.

Agility: Agile applications can incorporate updates, fixes, and new features quickly, ensuring that users face minimum service disruptions.

Responsiveness: Users expect personalized, real-time feedback from modern applications (e.g., personalized product recommendations in e-commerce apps). Responsiveness is essential for user satisfaction and engagement.

Intelligence: With the increasing volume of new data formats (unstructured, semi-structured, images, audio, geospatial, etc.), developers have quickly realized that relational databases lack the schema flexibility and scalability features needed to build modern applications using these new data formats.

This gap drove the invention of document-oriented databases, which use flexible schemas and native support for nested structures (like JSON) to accommodate evolving data requirements. Unlike rigid relational tables, document models enable modern applications to efficiently store and query semi-structured data types—including text, video, and sensor feeds—while providing horizontal scalability for distributed workloads.

```
{
  "_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
}
```

The document model and AI applications

The document model allows you to store embeddings alongside other document data (as just another format supported by BSON), providing greater flexibility for integrating LLM-based workflows into existing or new AI applications.

MongoDB's document model combines the capabilities of a document database with a specialized vector store, resulting in a simpler architecture, less overhead, and a unified experience.

Also, MongoDB's advanced querying capabilities allow developers to efficiently retrieve information across entire documents, including vector search with embeddings, full-text search across fields, and hybrid search by combining both approaches, offering faster querying and better accuracy.

Achieving the same functionality with a relational database or point solution requires complex data pipelines and synchronization that can introduce technical debt and unintended errors while slowing development teams.

As applications in the AI era generate larger volumes and different types of data, there's a growing demand for databases that can handle rapidly changing data models and scale dynamically. This trend is heightened by the rise of agentic workflows, which demand even greater flexibility in data structures. Traditional relational databases, with their rigid schemas, aren't well-suited to meet these new needs.

In contrast, document-model databases like MongoDB offer flexible schemas that can easily adapt to evolving data models. This adaptability makes them ideal for applications utilizing generative AI, LLMs, and agentic workflows, where data structures can change quickly. The combination of AI-driven development and document databases is transforming how we build applications. Now, flexibility is essential from the outset, not just an afterthought.

Other advantages of the document model

The document model also provides performance and scalability advantages:

1. A document is stored as a single object in MongoDB, requiring only a single read from memory or disk. On the other hand, RDBMS JOINs require multiple reads from multiple locations.

2. As documents are self-contained, distributing the database across multiple nodes (a process called sharding) becomes simpler and makes it much easier to achieve horizontal scalability on commodity hardware. The DBA no longer needs to worry about the performance penalty of executing cross-node JOINs to collect data from different tables.

Joining collections

Typically, it is best to take a denormalized data modeling approach for operational databases—the efficiency of reading or writing an entire record in a single operation outweighs any modest increase in storage requirements. However, there are examples where normalizing data can be beneficial, especially when data from multiple sources needs to be blended for analysis. This can be done using the $lookup stage in the MongoDB Aggregation Framework.

The Aggregation Framework is a data aggregation pipeline modeled on data processing pipelines. documents enter a multi-stage pipeline that transforms them into aggregated results. Each stage transforms the documents as they pass through.

The $lookup aggregation pipeline stage provides JOIN capabilities in MongoDB, supporting the equivalent of SQL subqueries and NON-EQUI joins.

For instance, in a shopping cart application, imagine you have an orders collection and a products collection. The $lookup operator can enrich your order data by using the product_id in each order to find the corresponding item in the products collection and embed its full details into that order.

MongoDB also offers the $graphLookup aggregation stage to recursively look up a set of documents with a specific defined relationship to a starting document. Developers can specify the maximum depth for the recursion and apply additional filters to only search nodes that meet specific query predicates. $graphLookup can recursively query within a single collection or across multiple collections.

Application	RDBMS Action	MongoDB Action
Create Product Record	INSERT to (n) tables (product description, price, manufacturer, etc.)	insert() into 1 document
Display Product Record	SELECT and JOIN (n) product tables	find() single document
Add Product Review	INSERT to “review” table, foreign key to the product record	insert() to “review” collection, reference to product document

Figure 6: Analyzing queries to design an optimum schema.

Defining the document schema

An application’s data access patterns should drive schema design, with a specific focus on:

- The read/write ratio of database operations and whether it is more important to optimize performance for one over the other.
- The types of queries and updates performed by the database.
- The lifecycle of the data and the growth rate of documents.

As a first step, the project team should document the operations performed on the application’s data, comparing:

1. How are these operations currently implemented by the relational database?
2. How could these operations be implemented in MongoDB?

Figure 6 (above) represents an example of this exercise.

This analysis helps to identify the ideal document schema and indexes for the application data and workload based on the queries and operations to be performed against it.

The project team can also analyze the RDBMS logs to identify the existing application’s most common queries. This analysis identifies the data that is most frequently accessed together and can, therefore, potentially be stored together within a single MongoDB document.

Modeling Relationships With Embedding and Referencing

Deciding when to embed a document or create a reference between separate documents in different collections is an application-specific consideration. However, some general considerations guide the decision during schema design.

Embedding

Data with a 1:1 or 1:many relationship (where the "many" objects always appear with or are viewed in the context of their parent documents) are natural candidates for embedding within a single document. The concept of data ownership and containment can also be modeled with embedding. Using the product data example above in Figure 6, product pricing—both current and historical—should be embedded within the product document since it is owned by and contained within that specific product. If the product is deleted, the pricing becomes irrelevant.

However, not all 1:1 and 1:many relationships are suitable for embedding in a single document. Referencing between documents in different collections should be used when:

- A document is frequently read but contains data that is rarely accessed. Embedding this data only increases the in-memory requirements (the working set) of the collection.
- One part of a document is frequently updated and constantly growing in size, while the remainder is relatively static.
- The combined document size would exceed MongoDB's 16 MB document limit. For example, consider a many:1 relationship, like a single product that has thousands of reviews. If you try to embed every review into the main product document, its total size could easily exceed MongoDB's 16 MB limit.

Referencing

Referencing can help address the challenges mentioned earlier and is commonly used when modeling many-to-many relationships. However, the application will need to perform follow-up queries to resolve these references. This may require additional round trips to the server or necessitate a "join" operation using MongoDB's $lookup aggregation pipeline stage.

Different design goals

Comparing these two design options—embedding sub-documents versus referencing between documents—highlights a fundamental difference between relational and document databases:

- The RDBMS optimizes data storage efficiency (as it was conceived at a time when storage was the most expensive component of the system).
- MongoDB's document model is optimized for how the application accesses data (as performance, developer time, and speed to market are now more important than storage volumes).

Schema Evolution and Its Impact on Schema Design

MongoDB's flexible schema provides a major advantage over relational databases. Developers can choose their desired level of schema structure and validation, from lightweight, dynamic schemas for rapid iteration to strict rules for governance as applications scale.

By contrast, relational databases require a rigid, predefined schema. Every column, data type, and relationship must be declared upfront. This works when requirements are static, but falls short in modern development, where change is constant and agility is critical.

Example: Enhancing a Restaurant Review App

Imagine you're building a reviews platform. Initially, users leave text-based reviews. Later, you want to support star ratings (e.g., 1-5 stars), especially for mobile users who prefer quick interactions.

In a relational database, adding this feature typically requires:

1. Creating a new ratings table.
2. Defining a foreign key relationship to reviews.
3. Updating your ORM mappings.
4. Writing data migration scripts (if necessary).
5. Coordinating a release with DBAs and DevOps.
6. Retesting impacted queries.
7. Carefully deploying the application and database changes.

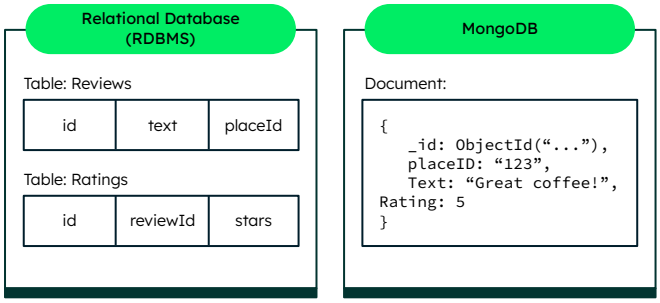
Even simple changes can trigger a long chain of updates and coordination.

In MongoDB, you simply:

1. Add a rating field to new documents in the reviews collection.
2. Adjust your application logic to read/write the new field.
3. Deploy the app.

Older documents remain valid. No data migration or schema lock-in. No downtime. No ALTER TABLE.

Adding a 'rating' to Reviews - RDBMS vs MongoDB



MongoDB's document model aligns with how developers build today—iteratively and rapidly. Schema changes are driven by the application, not blocked by the database. As projects mature, teams can layer on schema validation to ensure structure and consistency, without losing the flexibility that accelerates development.

Application Integration

After the schema is finalized, developers can easily connect their application to MongoDB using its rich ecosystem of language drivers. Database administrators can further configure the deployment for specific data consistency and durability requirements.

MongoDB Drivers and the API

Ease of use and developer productivity are two of MongoDB's core design goals.

MongoDB prioritizes developer productivity through an API that aligns with native programming language constructs. Where relational systems force data into rigid tables (creating an object-relational impedance mismatch), MongoDB's BSON-based document model maps directly to modern languages' object-oriented structures. Developers interact with data through native method calls (e.g., collection.insertOne() in JavaScript) that allow them to access data stored together without having to use JOINs. This coherence reduces cognitive load by eliminating translation layers between application code and database schema.

For example, an e-commerce "Order" with nested line items persists as a single document rather than fractured orders and order_items tables.

SQL Approach

```
-- Create tables for orders and items

CREATE TABLE orders (id INT, customer VARCHAR);

CREATE TABLE order_items (order_id INT, product VARCHAR, quantity INT);

-- Insert data into two tables

INSERT INTO orders VALUES (1, 'Maria');

INSERT INTO order_items VALUES (1, 'Coffee Maker', 2);
```

MongoDB Approach

```
Javascript

// Store everything in one document

db.orders.insertOne({

 order_id: 1,

customer: "Maria",

items: [

{ product: "Coffee Maker", quantity: 2 } ]});
```

MongoDB has idiomatic drivers for the most popular languages, including a dozen developed and supported by MongoDB (e.g., Java, JavaScript, Python, .NET, Go) and more than 30 community-supported drivers.

Developers can interact with MongoDB databases directly using the idioms and data structures of their chosen language.

With the MongoDB Atlas SQL Interface, you can leverage existing SQL knowledge and familiar tools to query and analyze Atlas data live. The Atlas SQL Interface uses mongosql, a SQL-92 compatible dialect that's designed for the document model. It also leverages Atlas Data Federation functionality under the hood so you can query across Atlas clusters and cloud storage, like S3 buckets.

Mapping SQL Database Objects to MongoDB Syntax

For developers familiar with SQL, it is useful to understand how core SQL statements such as CREATE, ALTER, INSERT, SELECT, UPDATE, and DELETE map to similar statements for MongoDB.

The comparison table in Figure 7 illustrates the difference between SQL and MongoDB concepts and semantics. MongoDB also offers an extensive array of advanced query operators.

Manually converting and validating SQL database objects like queries, triggers, and stored procedures is complex and time-consuming. MongoDB's Relational Migrator features a gen AI-powered Query Converter (Figure 8) that helps you avoid common pitfalls, like misusing $lookup. This feature helps you quickly convert SQL database objects into MongoDB equivalents, validate for compatibility, and generate development-ready code, significantly cutting migration and testing time. For example, Bendigo and Adelaide Bank completed its migration to MongoDB with 90% less human effort and at one-tenth of the cost of a traditional legacy migration using Relational Migrator and other MongoDB tools.

SQL Terms/Concepts	MongoDB Terms/Concepts
database	**database**
table	**collection**
row	**document** or **BSON** document
column	**field**
index	**index**
table joins	`$lookup`, embedded documents
primary key	**primary key**
aggregation (e.g. group by)	aggregation pipeline See the **SQL to Aggregation Mapping Chart.**
SELECT INTO NEW_TABLE	`$out` See the **SQL to Aggregation Mapping Chart.**
MERGE INTO TABLE	`$merge` See the **SQL to Aggregation Mapping Chart.**
UNION ALL	`$unionWith`
transactions	**transactions**

Figure 7: Difference between SQL and MongoDB terms.

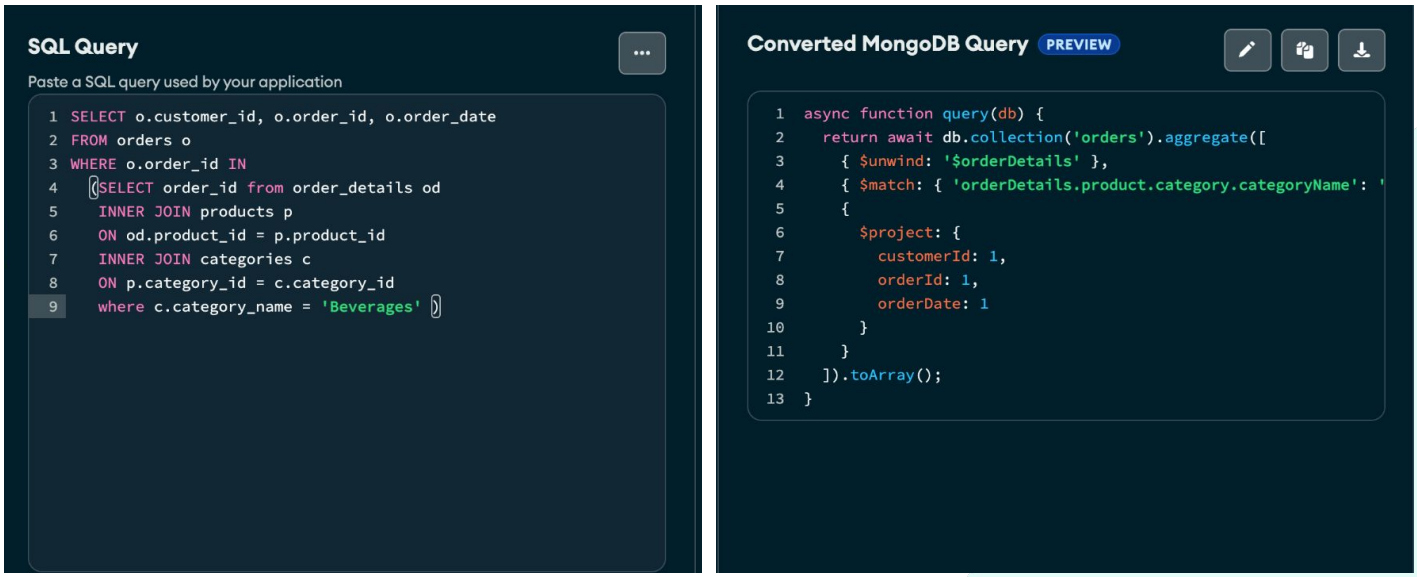


Figure 8: The Query Converter uses gen AI to convert SQL objects (queries, views, and stored procedures) to MongoDB syntax, simplifying and accelerating the migration process.

MongoDB Aggregation Framework

Aggregating data within any database is an important capability and a strength of the RDBMS. While many early NoSQL databases lacked robust aggregation capabilities, this is not universally true today. As a result, migrating to NoSQL databases historically forced developers to create workarounds, such as:

1. Building aggregations within their application code, increasing complexity, and compromising performance.

2. Exporting data to Hadoop or a data warehouse to run complex queries against the data. This also drastically increases complexity, duplicates data across multiple data stores, and does not allow for real-time analytics.

3. If available, writing native MapReduce operations within the NoSQL database itself.

MongoDB provides support for aggregation pipelines natively within the database, which offers functionality similar to SQL's GROUP BY, JOIN, and materialized view.

When using an aggregation pipeline, documents in a collection pass through a stepped process. Expressions produce output documents based on calculations performed on the input documents. The accumulator expressions used in the $group stage maintain state (e.g., totals, maximums, minimums, averages, standard deviations, and related data) as documents progress through the pipeline.

Additionally, an aggregation pipeline can manipulate and combine documents using projections, filters, redaction, lookups ('$lookup', which acts as JOIN), and recursive graph lookups ('$graphLookup'). It is also possible to transform data within the database—for example, using the $convert operator to cleanse data types into standardized formats.

The SQL to Aggregation Mapping Chart shows several examples demonstrating how MongoDB's Aggregation Framework handles SQL queries.

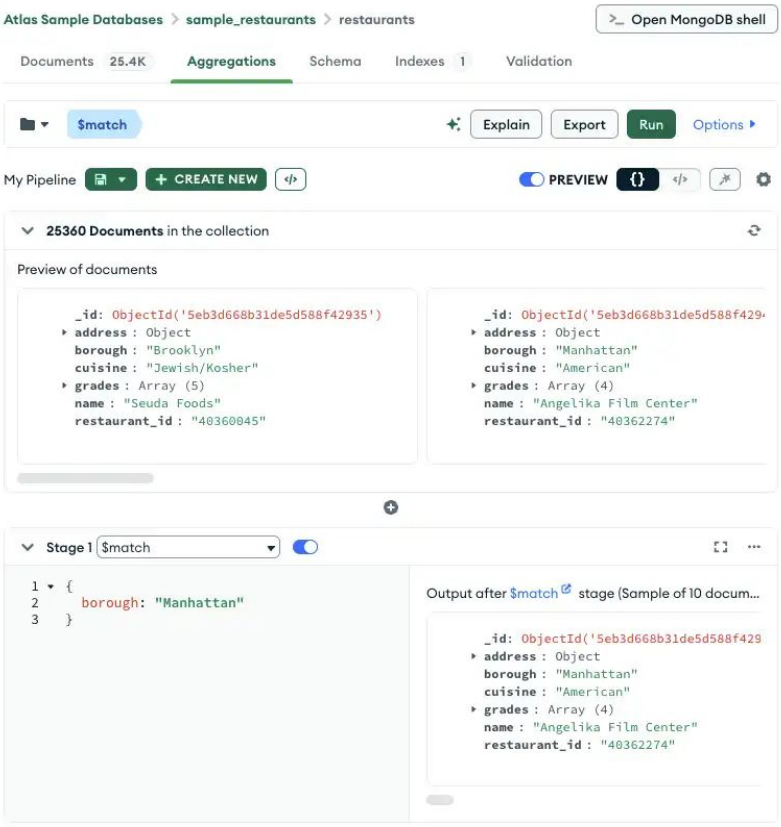


Figure 9: MongoDB Compass Aggregation Pipeline Builder.

You can also use the Aggregation Pipeline Builder in MongoDB Compass (Figure 9). You can write aggregation pipelines that allow documents in a collection or view to pass through multiple stages, where they are processed into a set of aggregated results.

Bonus tip: You can use the query bar in MongoDB Compass to ask natural language questions about your data.

Indexing

In any database, indexes are the single biggest tunable performance factor and are, therefore, integral to schema design. Indexes support the efficient execution of queries in MongoDB. If an appropriate index exists for a query, MongoDB uses the index to limit the number of documents it must scan.

Indexes in MongoDB largely correspond to indexes in a relational database. MongoDB uses B-tree indexes and natively supports secondary indexes. As such, the indexing paradigm will be immediately familiar to those coming from a SQL background.

The type and frequency of the application's queries should inform index selection. As with all databases, indexing does not come free: it imposes overhead on writes and resources (disk and memory).

Index types

MongoDB's rich query paradigm provides flexibility in accessing data. By default, MongoDB creates an index on the document's _id primary key field.

All user-defined indexes are secondary indexes. Indexes can be created in any part of the JSON document, including inside sub-documents and array elements. This makes them much more powerful than the indexes offered by relational databases.

Index options for MongoDB include:

- Single field indexes
- Compound indexes
- Unique indexes
- Multikey indexes
- Vector indexes with Atlas Vector Search
- Geospatial indexes
- Wildcard indexes
- Partial indexes
- Hashed indexes
- Text search indexes

ESR Rule

The ESR (Equality, Sort, Range) Rule in MongoDB is a guideline for designing efficient compound indexes. It improves MongoDB query performance by optimizing how compound indexes are structured to minimize data scanning, reduce memory usage, and accelerate sorting or filtering.

The ESR rule prioritizes index fields in this order:
Equality → Sort → Range

This sequence allows MongoDB to quickly narrow results with equality matches (e.g., status: "active"). Leverage the index for sorting without loading all data into memory and efficiently filter range queries (e.g., price > 100) on a reduced dataset.

Optimizing performance with indexes

MongoDB's query planner selects the index empirically by running alternate query plans and selecting the plan with the best response time.

The explain() method enables you to test queries from your application, showing information about how a query will be, or was, resolved.

You can also use MongoDB Compass—the GUI tool for MongoDB, to visualize the "explain" output, making it even easier to identify and resolve performance issues.

MongoDB offers a purpose-built tool to monitor and optimize slow queries. Performance Advisor monitors queries that MongoDB considers slow and suggests new indexes to improve query performance. The threshold for slow queries varies based on the average time of operations on your cluster to provide recommendations pertinent to your workload.

Queries are run against a sample collection to demonstrate the benefits of a recommended index. Each recommended index includes sample queries that are grouped by query shape. Performance Advisor doesn't negatively affect the performance of your Atlas clusters.

The Atlas Query Profiler helps diagnose and monitor slow-running queries using log data from your cluster. It aggregates, filters, and visualizes performance statistics in a scatter plot chart through the Query Insights interface. It is only available in M10+ and serverless. If you are running MongoDB on-premises, Ops Manager—part of MongoDB Enterprise Advanced—also includes a query profiler.

Figure 10 provides a high-level view of the information that makes it easy to quickly identify outliers and general trends. The table below offers operational statistics by namespace (database and collection) and operation type. You can choose which metric to filter and list operations. This includes operation execution time, documents scanned to returned ratio, whether an index was used, whether an in-memory sort occurred, and more. You can also select a specific time frame for the operations displayed, from the past 15 minutes up to the past 24 hours.

Once you have identified which operations are potentially problematic, the Query Profiler allows you to dig deeper into operation-level statistics to gain more insight into what's happening. You can view granular information on a specific operation in the context of similar operations, which can help you identify what general optimizations need to be made to improve performance.

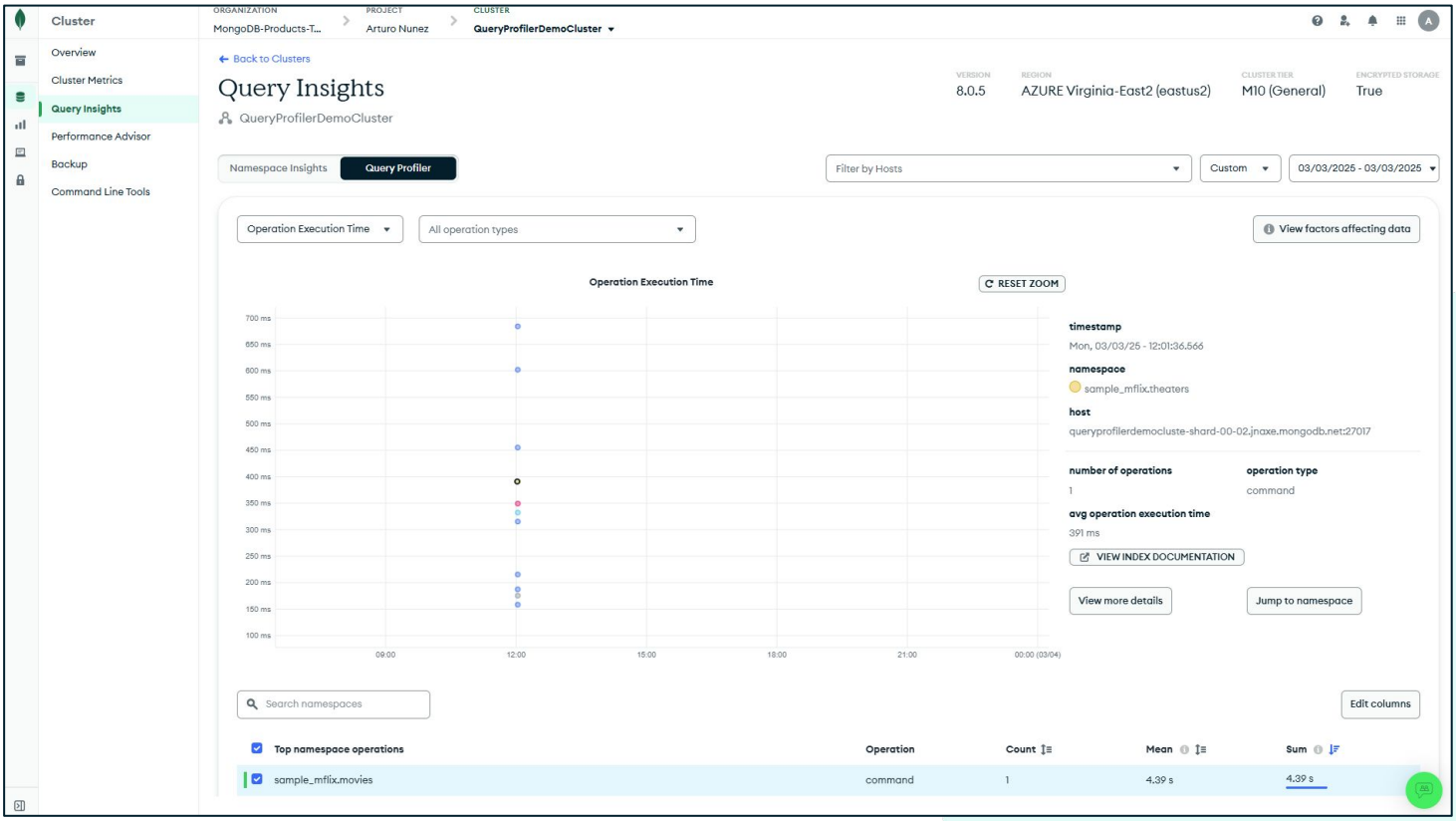


Figure 10: MongoDB Atlas Query Profiler.

Multidocument ACID Transactional Model

In MongoDB, related data is stored together in a single document. This means you can safely update multiple parts of that document in one atomic (all-or-nothing) operation. For most applications, this provides the same data integrity as a traditional database transaction.

If an update fails for any reason, the entire operation is rolled back, guaranteeing that your application will only ever see the complete and consistent version of the document.

MongoDB supports also multidocument ACID transactions and distributed transactions that operate across scaled-out, sharded clusters.

Multidocument transactions make it even easier for developers to address a complete range of use cases with MongoDB. They feel just like the transactions developers are familiar with from relational databases—multistatement, similar syntax, and easy to add to any application.

Through snapshot isolation, transactions provide a consistent view of data, enforce all-or-nothing execution, and do not impact performance for workloads that do not require them. For those operations that do require multidocument transactions, there are several best practices that developers should observe.

You can review all best practices in the MongoDB documentation for multidocument transactions.

Maintaining Strong Consistency

As a distributed system, MongoDB handles the complexity of maintaining multiple copies of data via replication. Read and write operations are directed to the primary replica by default for strong consistency, but users can choose to read from secondary replicas for reduced network latency, especially when users are geographically dispersed, or for isolating operational and analytical workloads running in a single cluster.

MongoDB provides tunable consistency to match your application's exact needs. It can enforce the strictest linearizable or causal consistency when required. For more flexibility, you can configure it to read data that has been committed to a majority of nodes (which can't be rolled back after a primary election) or even from just a single replica. This level of control allows MongoDB to satisfy the full range of consistency, performance, and geo-locality requirements of modern applications.

Write Durability

Write durability is crucial because it guarantees that once a transaction is successfully completed (committed), its changes are permanently stored and will survive any system failures, such as crashes, power outages, or hardware malfunctions.

Relational databases generally enforce strong write durability by default, ensuring every committed change is fully persistent and recoverable. However, this commitment to durability often sacrifices performance due to synchronous disk writes. While relational databases do offer some configuration to relax durability (e.g., per-transaction delayed durability in SQL Server and session-level settings in MySQL), they typically lack the per-operation granular control seen in MongoDB.

MongoDB provides highly granular write concerns, allowing you to fine-tune durability for each operation. Options range from "fire and forget" for maximum speed with minimal safety, to waiting for acknowledgments from multiple replicas and on-disk journaling for robust data persistence. This offers a level of operational control over durability that relational databases generally don't have.

If opting for the most relaxed write concern, the application can send a write operation to MongoDB and then continue processing additional requests without waiting for a response from the database, ensuring maximum performance. This option is useful for applications like logging, where users are typically analyzing trends in the data rather than discrete events.

With the default stronger write concerns, write operations wait until MongoDB applies and acknowledges the operation. The behavior can be further tightened by also opting to wait for replication of the write to:

- A single secondary.
- A majority of secondaries.
- A specified number of secondaries.
- All of the secondaries—even if they are deployed in different data centers (users should evaluate the impacts of network latency carefully in this scenario).

The write concern setting guarantees that a change is persisted to disk before the operation is acknowledged. It's configured through the driver and offers granular control—you can set it for a single operation, an entire collection, or the database as a whole.

Users can learn more about write concerns in our documentation.

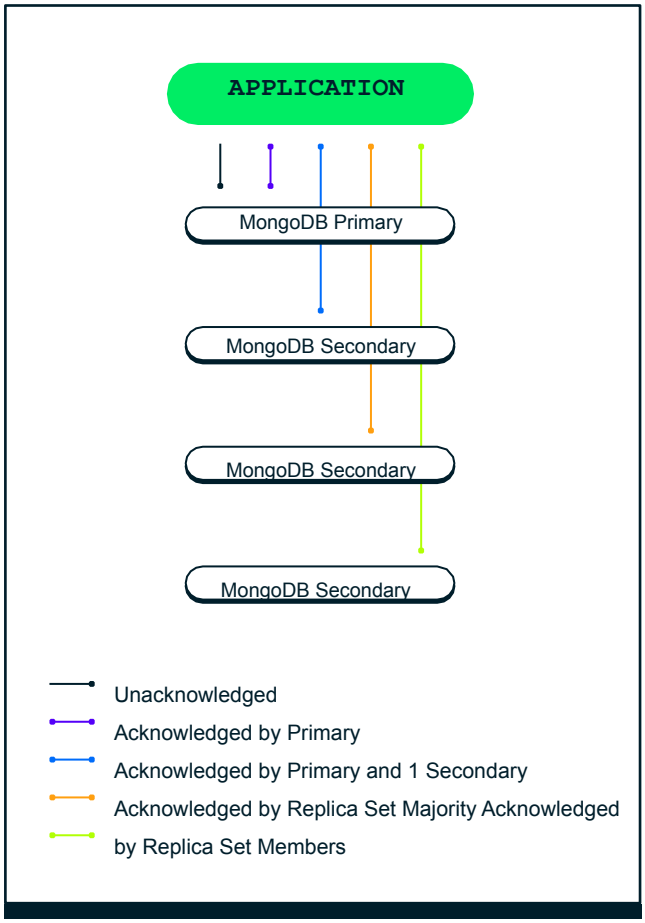


Figure 11: Configure durability per operation

MongoDB uses write-ahead logging to an on-disk journal to guarantee write operation durability and provide crash resilience. Before applying a change to the database—whether it is a write operation or an index modification—MongoDB writes the change operation to the journal. If a server failure occurs or MongoDB encounters an error before it can write the changes from the journal to the database, the journaled operation can be reapplied, thereby maintaining a consistent state when the server is recovered.

Implementing Validation and Constraints

Foreign keys

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

With the document model, referential integrity is built into 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.

Schema governance

While MongoDB's flexible schema is a powerful feature for many users, there are situations where strict guarantees on data structure and content are required. Using schema validation, developers and DBAs can define a prescribed document structure for each collection, which can reject any documents that do not conform to it. With schema validation, MongoDB enforces strict controls over JSON data:

- **Complete schema governance:** Administrators can define when additional fields are allowed to be added to a document, and specify a schema on array elements, including nested arrays.

- **Tunable controls:** Administrators have the flexibility to tune schema validation according to use case. For example, if a document fails to comply with the defined structure, it can either be rejected or still written to the collection while logging a warning message. The structure can be imposed on just a subset of fields (for example, requiring a valid customer name and address, while other fields can be freeform, such as social media handle and cell phone number). And of course, validation can be turned off entirely, allowing complete schema flexibility, which is especially useful during the development phase of the application.

- **Queryable:** The schema definition can be used by any query to inspect document structure and content. For example, DBAs can identify all documents that do not conform to a prescribed schema.

You can add a JSON schema to enforce these rules:

- Each document must contain a field named *lineItems.*
- The document may optionally contain other fields.
- *lineItems* must be an array where each element:
  - Must contain a *title* (string) and price (number no smaller than 0).
  - May optionally contain a boolean named *purchased.*
  - Must contain no further fields.

Bonus tip: As a developer, you can use Relational Migrator's Query Converter to learn MongoDB query syntax. This tool allows you to convert complex SQL queries into MongoDB Query API Syntax. Compare the two and understand their differences.

```
db.createCollection("orders", {
  validator: {
    $jsonSchema: {
      properties: {
        lineItems: {
          type: "array",
          items: {
            properties: {
              title: { type:
"string" },
              price: { type:
"number", minimum: 0.0 },
              purchased: { type:
"boolean" }
            },
            required: ["_id",
"title", "price"],
            additionalProperties:
false
          }
        }
      },
      required: ["lineItems"]
    }
  }
})
```

On-Demand Materialized Views

Materialized views let you pre-compute and store the results of common analytics queries—a valuable feature found in both relational databases and MongoDB. Typical use cases in MongoDB include:

- Rolling up a summary of sales data every 24 hours.
- Aggregating averages of sensor events every hour in an IoT app.
- Merging new batches of cleansed market trading data with a centralized MongoDB-based data warehouse so traders get refreshed market views across their portfolios.

Using the $merge stage, outputs from aggregation pipeline queries can be merged with existing stored result sets whenever you run the pipeline, enabling you to create materialized views that are refreshed on demand. Rather than a full-stop replacement of the existing collection's content, you can increment and enrich views of your result sets as new data is processed by the aggregation pipeline.

With MongoDB's materialized views, you have the flexibility to output results to sharded collections, enabling you to scale out your views as data volumes grow. You can also write the output to collections in different databases, further isolating operational and analytical workloads from one another. As the materialized views are stored in a regular MongoDB collection, you can apply indexes to each view, enabling you to optimize query access patterns and run deeper analysis against them using Atlas Charts or the BI and Apache Spark connectors.

On-demand materialized views represent a powerful addition to the analytics capabilities offered by MongoDB. They enable users to get faster insights from live, operational data without the expense and complexity of moving data through fragile ETL processes into dedicated data warehouses.

Operational Agility at Scale: MongoDB Atlas

MongoDB Atlas is a fully managed cloud database service that lets you deploy, run, and scale MongoDB databases without managing the underlying infrastructure. It is available in more than 125 regions across major cloud providers, including AWS, Azure, and Google Cloud.

Most companies have already moved to the public cloud to reduce the operational overhead of managing infrastructure and provide their teams with access to on-demand services that give them the agility they need to meet faster application development cycles. This move from building IT to consuming IT as a service is well aligned with parallel organizational shifts, including agile and DevOps methodologies and microservices architectures. Collectively, these seismic shifts in IT help companies prioritize developer agility, productivity, and time to market.

Getting started with MongoDB Atlas is simple. It uses a pay-as-you-go model with hourly billing, allowing you to deploy a cluster in minutes using the intuitive GUI or the Atlas CLI. Just select your cloud provider, region, and instance size, and Atlas handles the rest—automatically applying operational best practices so you can focus on your application instead of backend database management.

Here are some key features of MongoDB Atlas:

- Automated database and infrastructure provisioning, along with auto-scaling, so teams can get the database resources they need when they need them and scale elastically in response to application demands. Atlas is designed for distributed deployments, fault tolerance, and workload isolation.
- Native support for:
    - Atlas Search: Embedded full-text search in MongoDB Atlas that gives you a seamless, scalable experience for building relevance-based application features. Built on Apache Lucene, Atlas Search eliminates the need to run a separate search system alongside your database.
    - Atlas Vector Search: Enables you to perform searches on your data based on semantic meaning rather than just keywords. This helps you retrieve more relevant results and enables your AI-powered applications to support use cases such as semantic search, hybrid search, and generative search, including retrieval-augmented generation (RAG).
    - Atlas Stream Processing: Lets you process streams of complex data using the same Query API used in Atlas databases. It processes data instantly, checks for errors or delays in incoming data streams, and continuously publishes results to Atlas collections or Apache Kafka clusters, ensuring updated views and analysis of data at all times.
- Industry-leading security features to protect your data, with network isolation, fine-grained access control, auditing, and full-lifecycle data encryption: in transit, at rest, and—with our industry-leading Queryable Encryption technology—even while in use.
- Certifications against global standards, including ISO 27001, SOC 2, and more to help you achieve your compliance requirements. Atlas can also be used for workloads subject to regulatory standards such as HIPAA, PCI-DSS, and GDPR.
- Fully managed backups with point-in-time recovery to protect against data corruption and the ability to query backups in place without full restores.
- Fine-grained monitoring and customizable alerts for comprehensive performance visibility for developers and administrators.

- Automated patching and single-click upgrades for new major versions of the database, enabling you to take advantage of the latest MongoDB features.
- Integrated live migration tools to move your self-managed MongoDB clusters into the Atlas service or to move Atlas clusters between cloud providers.
- Built-in tools, alerts, charts, integrations, and logs to help you monitor your clusters.

MongoDB Atlas serves a wide range of workloads for startups, Fortune 500 companies, and government agencies, including mission-critical applications handling highly sensitive data in regulated industries. The developer experience across MongoDB Atlas and self-managed MongoDB is consistent, ensuring that you can easily move from on-premises to the public cloud and between providers as your needs evolve.

Migrating RDBMS Data With Relational Migrator

Data migration teams can use Relational Migrator, a free tool that leverages intelligent algorithms and generative AI to automate the most complex aspects of migrating from traditional relational databases to MongoDB. For large-scale and complex migrations, MongoDB's Professional Services team offers dedicated migration and application modernization services.

Overview of Relational Migrator's Key Features

Relational Migrator Feature	Benefit
Identify potential data and configuration risks before migrating to MongoDB	Pre-migration analysis automates the migration assessment process. It uses advanced algorithms to analyze the source database's schema, highlights potential data and configuration risks, and provides tailored recommendations to help you successfully migrate to MongoDB.
Easily map and customize your SQL schema to MongoDB	Build your schema from scratch or customize a recommended schema to meet your application requirements using the intuitive visual mapping feature.
Seamlessly migrate data from SQL to MongoDB	Migrate your mapped data to Atlas (cloud) or on-premises MongoDB using flexible migration options—a snapshot migration or continuous sync. Integrate Kafka for large-scale migrations. The tool supports migrations from several SQL databases, including Oracle, SQL Server, PostgreSQL, Sybase ASE, IBM Db2, CockroachDB, Yugabyte, and more.
Instantly generate development-ready application code	Generate development-ready code for entity classes, persistence layers, and APIs in C#, Java, JavaScript, and JSON, eliminating the need for manual code rewriting.
Convert SQL database objects with AI and validate them	With generative AI, you can convert SQL queries, views, and stored procedures to MongoDB code and validate them to ensure compatibility. The converter supports C#, Java, and JavaScript.

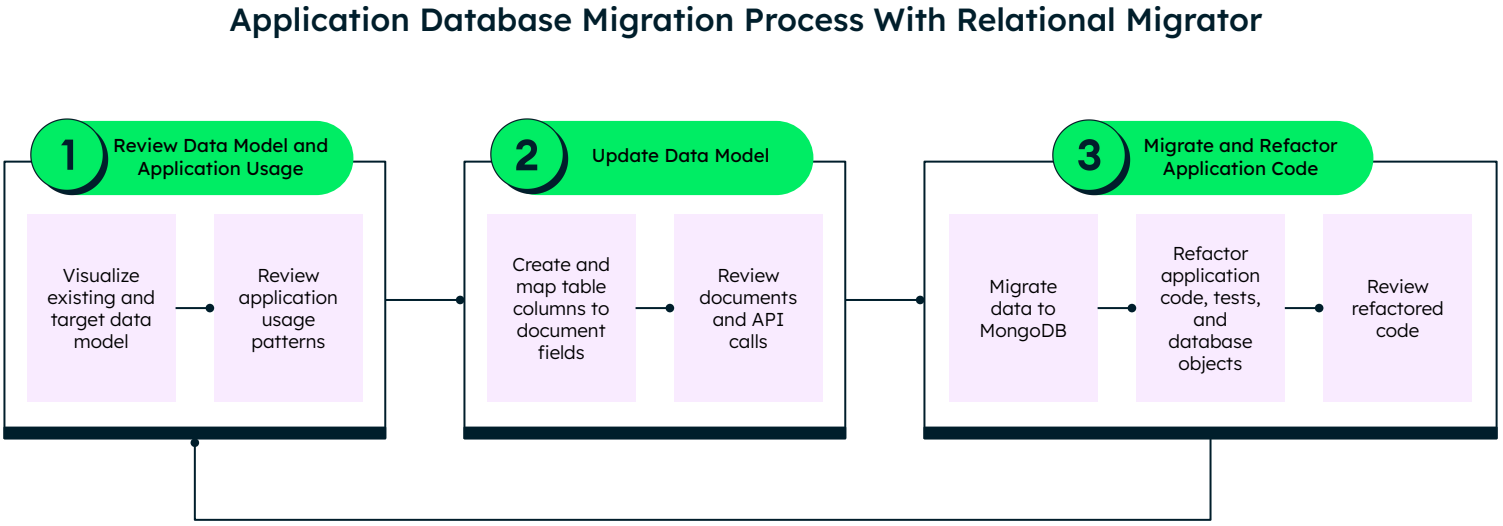


Figure 12: The relational database migration process in Relational Migrator.

Bonus tip: Refer to this MySQL to MongoDB migration guide to see the step-by-step process of migrating MySQL database to MongoDB Atlas using Relational Migrator.

Migration Support: MongoDB Professional Services

MongoDB's global Professional Services team is here to support you with your migration initiatives, regardless of how far along you are in the process. Whether you're seeking expert advice on which applications to migrate, need help creating a migration plan and roadmap, want to collaborate with specialists to oversee your migration, or require a complete software development team to execute the migration, our team is dedicated to ensuring your successful transition to MongoDB.

Learn more about how to successfully migrate to MongoDB with the help of our experts.

Enabling Your Teams: MongoDB University and Training

Regardless of how you migrate, making sure your teams are skilled in working with MongoDB is critical for long-term success. Upskilling your development teams not only ensures a smooth transition but also fosters ongoing innovation, allowing you to fully leverage the capabilities of MongoDB.

To help with this, MongoDB offers free self-paced courses and instructor-led training to boost technical expertise.

- MongoDB University offers free on-demand online courses for all skill levels to help users learn MongoDB.
- Instructor-led training provides hands-on, live training sessions for all skill levels, delivered in private or public sessions.

Attend a .local MongoDB Event

MongoDB.local connects you to MongoDB experts and users in a city near you! Meet our experts building the products and users shaking up their industries. Find a local event near you on our events page.

Conclusion

Following the best practices outlined in this guide can help project teams reduce the time and risk of database migrations, while enabling you to take advantage of the benefits of MongoDB and the document model. In doing so, you can quickly start to realize a more agile, scalable, and cost-effective infrastructure, innovating on applications that were never before possible.

We Can Help

We are the company that builds and runs MongoDB. Over 54,500 organizations rely on our commercial products. We offer software and services to make your life easier:

- MongoDB Atlas is a fully-managed cloud database service that lets you deploy and scale MongoDB without managing infrastructure. It offers features like auto-scaling, backups, and performance optimization, and is available in over 125 regions on AWS, Azure, and Google Cloud.
- 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 Compass is a free interactive GUI tool for querying, optimizing, and analyzing your MongoDB data. Get key insights, drag and drop to build pipelines, and more.
- MongoDB Relational Migrator is a free tool that leverages intelligent algorithms and generative AI to automate the most complex aspects of migrating from traditional relational databases to MongoDB, significantly reducing the time and effort required for migration while minimizing associated risks.
- MongoDB Professional Services helps you accelerate your most important modernization initiatives. Arm your teams with the deepest bench of MongoDB expertise, enabling them to drive success and reduce risk at every stage of your digital transformation and application development journey.
- MongoDB University 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

Case Studies

Free Online Training

Events and Webinars

Documentation

MongoDB.