A MongoDB White Paper

RDBMS to MongoDB Migration Guide

Considerations and Best Practices
February 2020

mongoDB

Table of Contents

Introduction

The relational database has been the foundation of enterprise data management for forty years. But the way we build and run applications today, coupled with unrelenting growth in new data sources and user loads are 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.

Around 30% of all MongoDB projects are now migrations from relational databases. MongoDB is designed to meet the demands of modern apps with a technology foundation that enables you through:

1. The document data model – presenting you **the best way to work with data**.
2. A distributed systems design – allowing you to **intelligently put data where you want it**.
3. A unified experience that gives you the **freedom to run anywhere** – allowing you to future-proof your work and eliminate vendor lock-in.

As illustrated in Figure 1, enterprises from a variety of industries have migrated successfully from relational database management systems (RDBMS) to MongoDB for myriad applications.

This guide is designed for project teams that want to know how to migrate from an RDBMS to MongoDB. We provide a step-by-step roadmap, depicted in Figure 2.

Many links are provided throughout this document to help guide users to the appropriate resources online. For the most current and detailed information on particular topics, please see the online documentation.

Organizing for Success: Application Modernization Factory

Many users have successfully executed migrations from relational databases to MongoDB using their own internal resources. In addition, MongoDB has worked with many companies to support larger scale, more strategic

Organization	Migrated From	Application
Cisco	Commerical RDBMS	eCommerce Platform
eHarmony	Oracle & Postgres	Customer Data Management & Analytics
Shutterfly	Oracle	Web and Mobile Services
Sega	MySQL	Gaming Platforms
Under Armour	Microsoft SQL Server	eCommerce
Baidu	MySQL	100+ Web & Mobile Service
MTV Networks	Multiple RDBMS	Centralized Content Management
Telefonica	Oracle	Customer Account Management
Verizon	Oracle	Single View, Employee Systems
The Weather Channel	Oracle & MySQL	Mobile Networking Platforms

Figure 1: Case Studies

migrations of their application estate. The Application Modernization Factory (AMF) is a professional services engagement that provides advisory consulting, program governance, and application lifecycle expertise.

Working with MongoDB consultants, the first step in the AMF process is to identify application stakeholders, and then build an inventory and characterization of existing apps, before identifying the best-fit candidates for modernization. From there, we scope the project, quantify the economic value of change, and provide a roadmap for delivery.

We support the modernization of applications throughout the software development lifecycle, harnessing patterns and technologies such as agile and DevOps, microservices, cloud computing, and MongoDB best practices. We partner with your teams to accelerate the assessment, prioritization, and redesign of legacy apps, and work with them through the modernization efforts of redevelopment, consolidation, and optimization. To learn more, review the MongoDB Legacy Modernization page.

Getting Started: Schema Design

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 apply to most schema migration projects.

Before exploring schema design, Figure 3 provides a useful reference for translating terminology from the relational to MongoDB world.

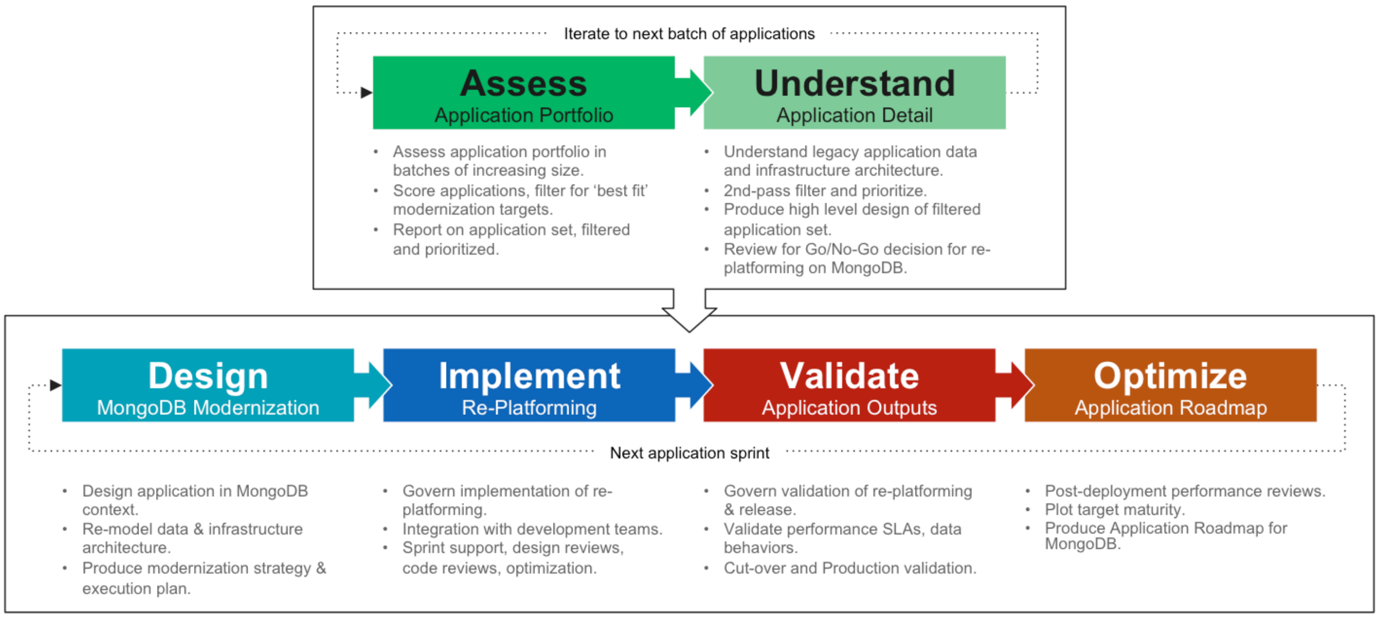

Iterate to next batch of applications
Assess
Application Portfolio
Understand
Application Detail
Assess application portfolio in batches of increasing size.
Score applications, filter for 'best fit' modernization targets.
Report on application set, filtered and prioritized.
Understand legacy application data and infrastructure architecture.
2nd-pass filter and prioritize.
Produce high level design of filtered application set.
Review for Go/No-Go decision for re-platforming on MongoDB.
Design
MongoDB Modernization
Implement
Re-Platforming
Validate
Application Outputs
Optimize
Application Roadmap
Next application sprint
Design application in MongoDB context.
Re-model data & infrastructure architecture.
Produce modernization strategy & execution plan.
Govern implementation of re-platforming.
Integration with development teams.
Sprint support, design reviews, code reviews, optimization.
Govern validation of re-platforming & release.
Validate performance SLAs, data behaviors.
Cut-over and Production validation.
Post-deployment performance reviews.
Plot target maturity.
Produce Application Roadmap for MongoDB.

Figure 2: MongoDB Application Modernization Factory accelerating and de-risking RDBMS Migration

RDBMS	MongoDB
Database	Database
Table	Collection
Row	Document
Column	Field
Index	Index
JOIN	Embedded document, document references, or `$lookup` to combine data from different collections
Materialized View	On-demand Materialized View
ACID Transactions	Multi-document ACID Transactions that can be distributed across scale-out clusters

Figure 3: Terminology Translation

Schema design requires a change in perspective for data architects, developers, and DBAs:

- From the legacy relational data model that flattens data into rigid 2-dimensional tabular structures of rows and columns
- To a rich and dynamic document data model with embedded sub-documents and arrays

From Rigid Tables to Flexible and Dynamic BSON Documents

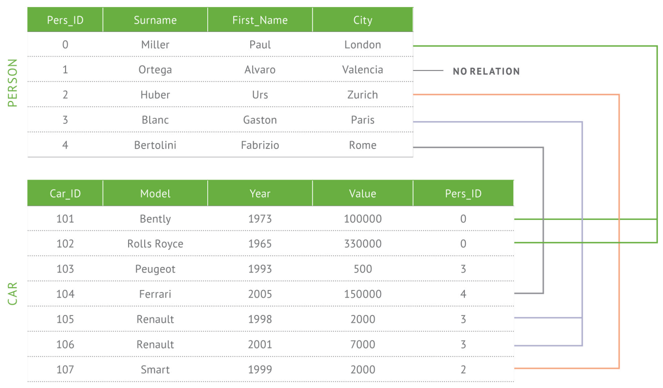
Much of the data we use today has complex structures that can be modeled and represented more efficiently using JSON (JavaScript Object Notation) documents, rather than tables.

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 int, decimal, long, and floating point.

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

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

The project team should start the schema design process by considering the application's requirements. It should model the data in a way that takes advantage of the document model's flexibility. 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 would commonly be collapsed (embedded) into a single document in MongoDB.


PERSON
NO RELATION
CAR

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

Car_ID	Model	Year	Value	Pers_ID
101	Bently	1973	100000	0
102	Rolls Royce	1965	330000	0
103	Peugeot	1993	500	3
104	Ferrari	2005	150000	4
105	Renault	1998	2000	3
106	Renault	2001	7000	3
107	Smart	1999	2000	2

Figure 4: Relational Schema, Flat 2-D Tables

In Figure 4, the RDBMS uses the `Pers_ID` field to JOIN the `Person` table with the `Car` table to enable the application to report each car's owner. Using the document model, embedded sub-documents and arrays effectively pre-JOIN data by combining related fields in a single data structure. Rows and columns that were traditionally normalized and distributed across separate tables can now be stored together in a single document, eliminating the need to JOIN separate tables when the application has to retrieve complete records.

Modeling the same data in MongoDB enables us to create a schema in which we embed an array of sub-documents for each car directly within the Person document.

```
{
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, ….},
]
}
```

In this simple example, the relational model consists of only two tables. (In reality most applications will need tens, hundreds or even thousands of tables.) This approach does not reflect the way architects think about data, nor the way in which developers write applications.

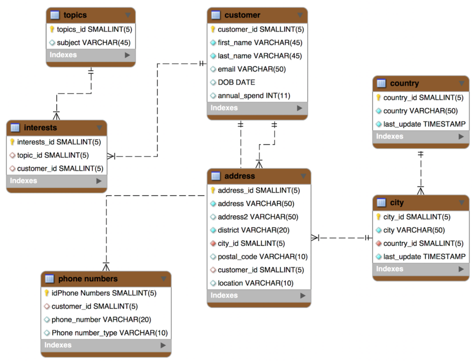


Figure 5: Modeling a customer with the relational database: data is split across multiple tables

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.

```
{
"_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
}
```

Documents are a much more natural way to describe data. This allows documents to be closely aligned to the structure of objects in the programming language. As a result, it's simpler and faster for developers to model how data in the application will map to data stored in the database.

Other Advantages of the Document Model

In addition to making it more natural to represent data at the database level, the document model also provides performance and scalability advantages:

- The complete document can be accessed with a single call to the database, rather than having to JOIN multiple tables to respond to a query. The MongoDB document is physically stored as a single object, requiring only a single read from memory or disk. On the other hand, RDBMS JOINs require multiple reads from multiple physical locations.
- As documents are self-contained, distributing the database across multiple nodes (a process called sharding) becomes simpler and makes it possible to achieve horizontal scalability on commodity hardware. The DBA no longer needs to worry about the performance penalty of executing cross-node JOINs (should they even be possible in the existing RDBMS) to collect data from different tables.

Joining Collections

Typically it is most advantageous to take a denormalized data modeling approach for operational databases – the efficiency of reading or writing an entire record in a single operation outweighing 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 pipeline for data aggregation modeled on the concept of data processing pipelines. Documents enter a multi-stage pipeline that transforms the documents into aggregated results. The pipeline consists of stages; 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-equijoins.

As an example, if the left collection contains `order` documents from a shopping cart application then the `$lookup` operator can match the `product_id` references from those documents to embed the matching product details from the `products` collection.

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

Defining the Document Schema

The 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

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

Figure 6: Analyzing Queries to Design the Optimum Schema

for one over the other

- The types of queries and updates performed by the database
- The life-cycle of the data and growth rate of documents

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

1. How these are currently implemented by the relational database
2. How MongoDB could implement them

Figure 6 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 identify the existing application's most common queries by analyzing the logs maintained by the RDBMS. 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 instead create a reference between separate documents in different collections is an application-specific consideration. There are, however, some general considerations to 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, 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 of the document is relatively static.
- The combined document size would exceed MongoDB's 16MB document limit, for example when modeling many:1 relationships, such as product reviews to product.

Referencing

Referencing can help address the challenges cited above, and is also typically used when modeling many:many relationships. However the application will need to issue follow-up queries to resolve the reference, requiring additional round-trips to the server, or require a "joining" 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).

Data modeling is an expansive topic. To help you make the right decisions, here is a summary of the key resources you should review:

- The MongoDB documentation provides an extensive section on data modeling, starting from high level concepts of the document data model before progressing to practical examples and design patterns, including more detail on referencing and embedding.
- You should also review our Building with Patterns blog series to learn more about specific schema design best practices for different use cases, including catalog and content management, IoT, mobile apps, analytics, and single view (i.e. customer 360). It overlays these use cases with specific schema design patterns such as versioning, bucketing, referencing, and graphs.
- MongoDB University offers a no-cost, web-based training course on data modeling. This is a great way to kick-start your learning on schema design with the document data model.

Indexing

In any database, indexes are the single biggest tunable performance factor and are therefore integral to schema design.

Indexes in MongoDB largely correspond to indexes in a relational database. MongoDB uses B-Tree indexes, and natively supports secondary indexes. As such, it 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 resource (disk and memory) usage.

Index Types

MongoDB has a rich query model that provides flexibility in how data is accessed. 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 on any part of the JSON document – including inside sub-documents and array elements – making them much more powerful than those offered by relational databases.

Index options for MongoDB include:

- **Compound Indexes.** A single index structure that maintains references to multiple fields. For example, consider an application that stores data about customers. The application may need to find customers based on last name, first name, and state of residence. With a compound index on last name, first name, and state of residence, queries could efficiently locate people with all three of these values specified. An additional benefit of a compound index is that any leading field(s) within the index can be used, so fewer indexes on single fields may be necessary: this compound index would also optimize queries looking for customers by last name or last name and first name.
- **Unique Indexes.** By specifying an index as unique, MongoDB will reject inserts of new documents or updates to existing documents which would have resulted in duplicate values for the indexed field. If a compound index is specified as unique, the combination of values must be unique.
- **Array Indexes.** For fields that contain an array, each array value is stored as a separate index entry. For example, documents that describe a product might include an array containing its main attributes. If there is an index on the attributes field, each attribute is indexed and queries on the attribute field can be optimized by this index. There is no special syntax required for creating array indexes – if the field contains an array, it will be indexed as an array index.

- **TTL Indexes.** In some cases data should expire automatically. Time to Live (TTL) indexes allow the user to specify a period of time after which the database will automatically delete the data. A common use of TTL indexes is applications that maintain a rolling window of history (e.g., most recent 100 days) for user actions such as clickstreams.
- **Geospatial Indexes.** MongoDB provides geospatial indexes to optimize queries related to location within a two-dimensional space, such as projection systems for the earth. These indexes allow MongoDB to optimize queries for documents that contain a polygon or points that are closest to a given point or line; that are within a circle, rectangle or polygon; or that intersect a circle, rectangle or polygon.
- **Wildcard Indexes.** For workloads with many ad-hoc query patterns or that handle highly polymorphic document structures, wildcard indexes give you a lot of extra flexibility. You can define a filter that automatically indexes all matching fields, subdocuments, and arrays in a collection. As with any index, they also need to be stored and maintained, so will add overhead to the database. If your application's query patterns are known in advance, then you should use more selective indexes on the specific fields accessed by the queries.
- **Partial Indexes.** Partial Indexes can be viewed as a more flexible evolution of Sparse Indexes, where the DBA can specify an expression that will be checked to determine whether a document should be included in a particular index. e.g. for an "orders" collection, an index on state and delivery company might only be needed for active orders and so the index could be made conditional on `{orderState: "active"}` – thereby reducing the impact to memory, storage, and write performance while still optimizing searches over the active orders.
- **Hash Indexes.** Hash indexes compute a hash of the value of a field and index the hashed value. The primary use of this index is to enable hash-based sharding, a simple and uniform distribution of documents across shards.
- **Text Search Indexes.** MongoDB provides a specialized index for text search that uses advanced, language-specific linguistic rules for stemming, tokenization and stop words. Queries that use the text search index return documents in relevance order. Each collection may have at most one text index but it may include multiple fields. If you are running MongoDB in the fully-managed Atlas global cloud database service, consider using Atlas Search which provides a Lucene index integrated with the MongoDB database. FTS provides higher performance and greater flexibility to filter, rank, and sort through your database to quickly surface the most relevant results to your users.

Optimizing Performance With Indexes

MongoDB's query optimizer selects the index empirically by occasionally running alternate query plans and selecting the plan with the best response time. The query optimizer can be overridden using the `cursor.hint()` method.

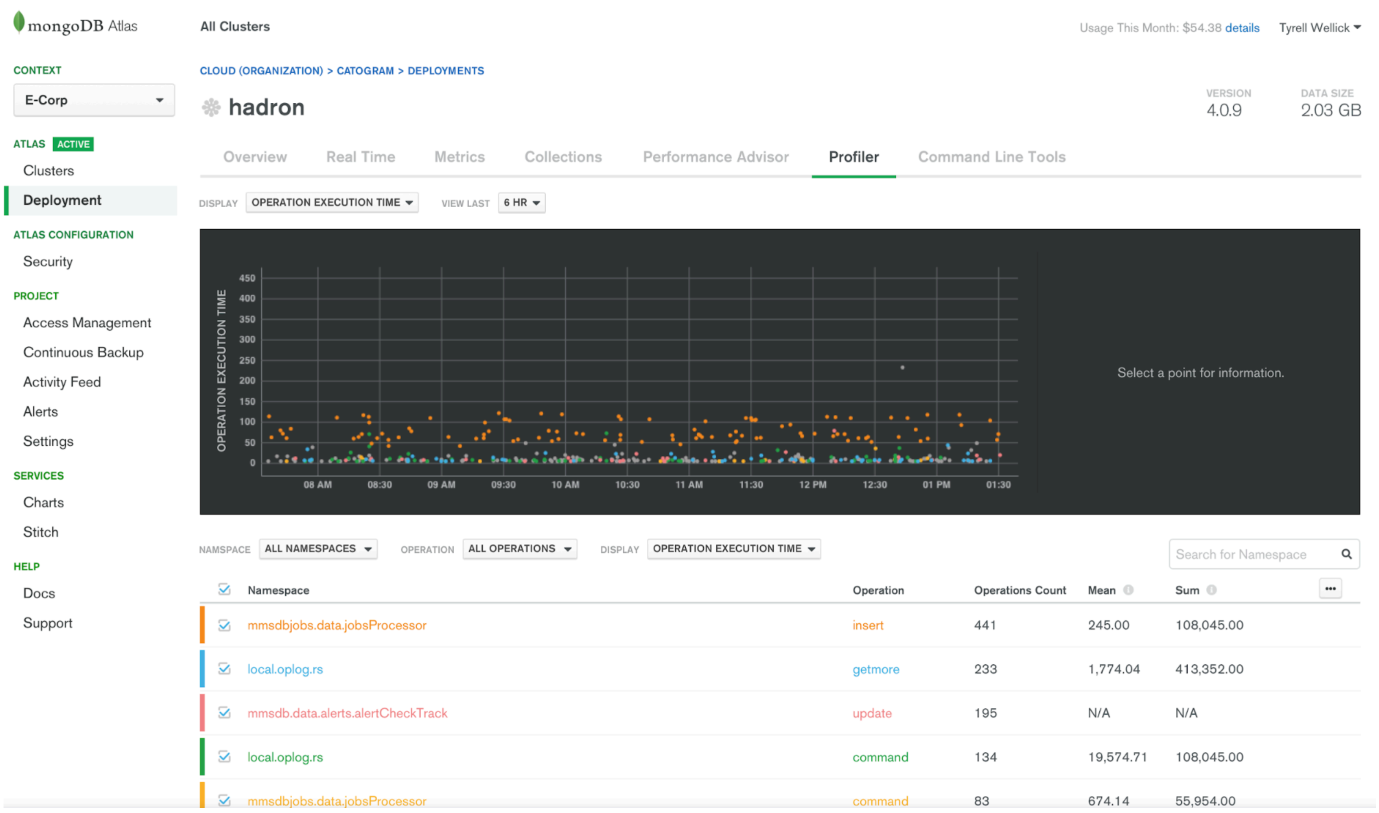
MongoDB's explain() method enables you to test queries from your application, showing information about how a query will be, or was, resolved, including:

- Which indexes were used
- Whether the query was covered by the index or not
- Whether an in-memory sort was performed, which indicates an index would be beneficial
- The number of index entries scanned
- The number of documents returned, and the number read
- How long the query took to resolve in milliseconds
- Which alternative query plans were rejected (when using the `allPlansExecution` mode)

The explain plan will show 0 milliseconds if the query was resolved in less than 1 ms, which is typical for well-tuned queries.

The MongoDB Compass GUI visualizes explain output, making it even easier for you to identify and resolve performance issues.

The MongoDB Query Profiler helps expose performance issues by displaying slow-running queries (by default, queries that exceed 100ms) and their key performance statistics directly in the UI.



VERSION 4.0.9
DATA SIZE 2.03 GB

DISPLAY: OPERATION EXECUTION TIME
VIEW LAST: 6 HR

NAMSPACE: ALL NAMESPACES
OPERATION: ALL OPERATIONS
DISPLAY: OPERATION EXECUTION TIME

Namespace	Operation	Operations Count	Mean	Sum
mmsdbjobs.data.jobsProcessor	insert	441	245.00	108,045.00
local.oplog.rs	getmore	233	1,774.04	413,352.00
mmsdb.data.alerts.alertCheckTrack	update	195	N/A	N/A
local.oplog.rs	command	134	19,574.71	108,045.00
mmsdbjobs.data.jobsProcessor	command	83	674.14	55,954.00

Figure 7: MongoDB Atlas Query Profiling

A chart provides a high-level view of that information that makes it easy to quickly identify outliers and general trends, while a table offers operation 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 select a specific time frame for the operations displayed, from the past 15 minutes 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. The MongoDB Atlas Query Profiler is available without additional cost or performance overhead.

If you are running MongoDB on-premises, Ops Manager – part of MongoDB Enterprise Advanced – also includes a query profiler.

Even with all of the telemetry provided by MongoDB's tools, you are still responsible for pulling and analyzing the required data to make decisions on which indexes to add.

MongoDB Atlas and Ops Manager eliminates this effort with the Performance Advisor which monitors queries that took more than 100ms to execute, and automatically suggests new indexes to improve performance.

Recommended indexes are accompanied by sample queries, grouped by query shape (i.e., queries with a similar predicate structure, sort and projection), that were run against a collection that would benefit from the addition of suggested index. The Performance Advisor does not negatively affect the performance of your Atlas clusters.

If you are happy with the recommendation, you can then roll out the new indexes automatically, without incurring any application downtime.

For all performance best practices, covering schema design, indexing, profiling, sharding, sizing and more, check out the MongoDB performance best practices blog series.

Indexes and Scaling-Out

While it may not be necessary to shard (partition) the database at the outset of the project, it is always good practice to assume that future scalability will be necessary (e.g., due to data growth or the popularity of the application). Defining index keys during the schema design phase also helps identify keys that can be used when implementing MongoDB's native sharding for application-transparent scale-out.

Schema Evolution and the Impact on Schema Design

MongoDB's dynamic schema provides a major advantage versus relational databases.

Collections can be created without first defining their structure, i.e., document fields and their data types. Documents in a given collection need not all have the same set of fields. One can change the structure of documents just by adding new fields or deleting existing ones.

Consider the example of a customer record:

- Some customers will have multiple office locations and lines of business, and some will not.
- The number of contact methods within each customer can be different.
- The information stored for each of these contact methods can vary. For instance, some may have public social media feeds which could be useful to monitor, and some will not.
- Each customer may buy or subscribe to different services from their vendor, each with their own sets of contracts.

Modeling this real-world variance in the rigid, two-dimensional schema of a relational database is complex and convoluted. In MongoDB, supporting variance between documents is a fundamental, seamless feature of BSON documents.

MongoDB's flexible and dynamic schemas mean that schema development and ongoing evolution are straightforward. For example, the developer and DBA working on a new development project using a relational database must first start by specifying the database schema, before any code is written. At a minimum this will take days; it often takes weeks or months.

MongoDB enables developers to evolve the schema through an iterative and agile approach. Developers can start writing code and persist the objects as they are created. And when they add more features, MongoDB will continue to store the updated objects without the need for performing costly `ALTER TABLE` operations or re-designing the schema from scratch.

These benefits also extend to maintaining the application in production. When using a relational database, an application upgrade may require the DBA to add or modify fields in the database. These changes require planning across development, DBAs, and operations teams to synchronize application and database upgrades, agreeing on when to schedule the necessary `ALTER TABLE` operations. Even trivial changes to an existing relational data model result in a complex dependency chain – from updating ORM class-table mappings to programming language classes that have to be recompiled and code changed accordingly.

As MongoDB allows schemas to evolve dynamically, such operations requires upgrading just the application, with typically no action required for MongoDB. Evolving applications is simple, and project teams can improve agility and time to market.

At the point that the DBA or developer determines that some constraints should be enforced on the document structure, Schema Validation rules can be added – further details are provided later in this guide.

Application Integration

With the schema designed, the project can move towards integrating the application with the database using MongoDB drivers and tools.

DBA's can also configure MongoDB to meet the application's requirements for data consistency and durability. Each of these areas are discussed below.

MongoDB Drivers and the API

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

One fundamental difference between a SQL-based RDBMS and MongoDB is that the MongoDB interface is implemented as methods (or functions) within the API of a specific programming language, as opposed to a completely separate text-based language like SQL. This, coupled with the affinity between MongoDB's BSON document model and the data structures used in modern programming languages, makes application integration simple.

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.

MongoDB's idiomatic drivers minimize onboarding time for new developers and simplify application development. For instance, Java developers can simply code against MongoDB natively in Java; likewise for Python developers, Go developers, and so on. The drivers are created by development teams that are experts in their given language and know how programmers prefer to work within those languages.

Mapping SQL 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 the MongoDB API. The documentation includes a comparison chart with examples to assist in the transition to MongoDB Query Language structure and semantics. In addition, MongoDB offers an extensive array of advanced query operators.

MongoDB Aggregation Pipeline

Aggregating data within any database is an important capability and a strength of the RDBMS. Many NoSQL databases do not have aggregation capabilities. As a result, migrating to NoSQL databases has traditionally forced developers to develop 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 the Aggregation Pipeline natively within the database, which delivers similar functionality to the `GROUP BY`, `JOIN`, Materialized View, and related SQL features.

When using the aggregation pipeline, documents in a collection pass through a processing, where they are processed in stages. 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, the aggregation pipeline can manipulate and combine documents using projections, filters, redaction, lookups (JOINs), and recursive graph lookups. 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 a number of examples demonstrating how queries in SQL are handled in MongoDB's aggregation pipeline.

Business Intelligence Integration – MongoDB Connector for BI

Driven by growing requirements for self-service analytics, faster discovery and prediction based on real-time operational data, and the need to integrate multi-structured and streaming data sets, BI and analytics platforms are one of the fastest growing software markets.

To address these requirements, modern application data stored in MongoDB can be easily explored with industry-standard SQL-based BI and analytics platforms. Using the BI Connector, analysts, data scientists, and

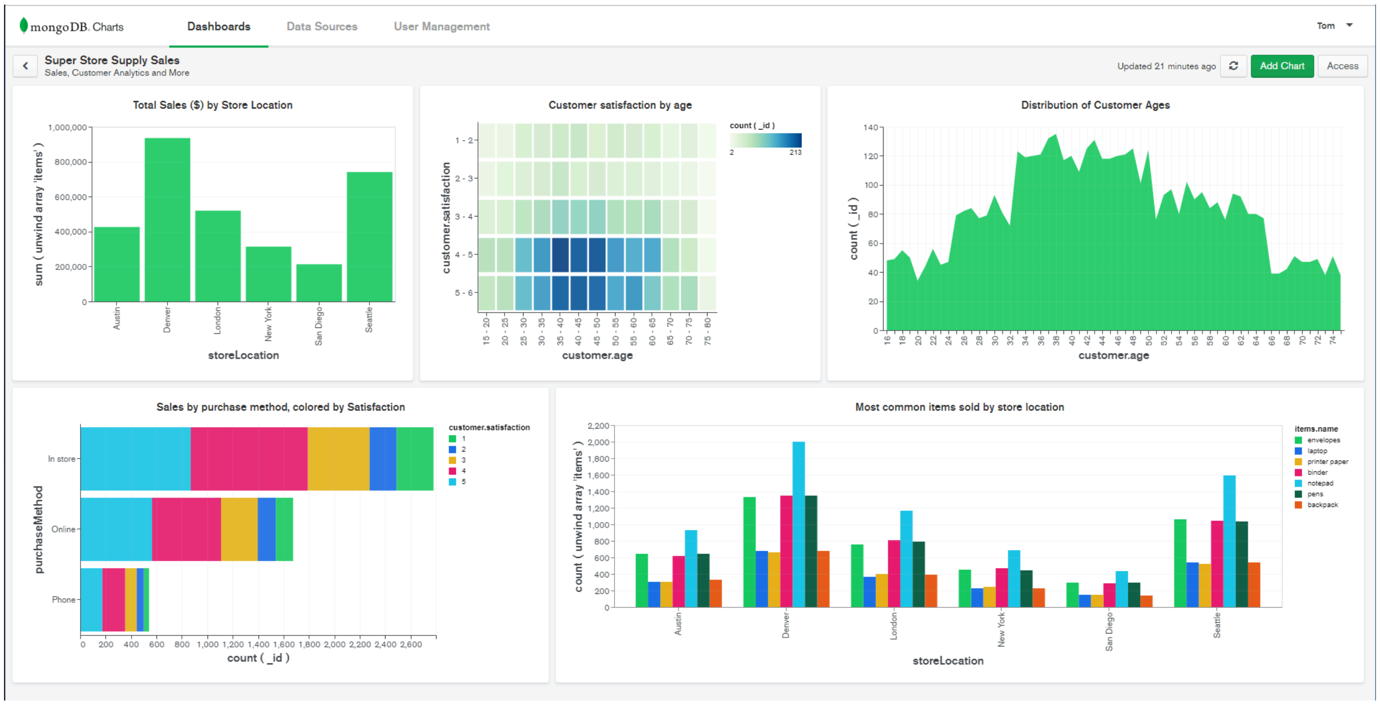

mongoDB. Charts
Dashboards
Data Sources
User Management
Tom
Super Store Supply Sales
Sales, Customer Analytics and More
Updated 21 minutes ago
Add Chart
Access
Total Sales ($) by Store Location
Customer satisfaction by age
Distribution of Customer Ages
Sales by purchase method, colored by Satisfaction
Most common items sold by store location
storeLocation
customer.age
purchaseMethod
count (_id)

Figure 8: Creating rich graphs and dashboards directly from the Charts UI

business users can seamlessly visualize semi-structured and unstructured data managed in MongoDB, alongside traditional data in their SQL databases, using the same BI tools deployed within millions of enterprises.

SQL-based BI tools such as Tableau expect to connect to a data source with a fixed schema presenting tabular data. This presents a challenge when working with MongoDB's dynamic schema and rich, multi-dimensional documents. In order for BI tools to query MongoDB as a data source, the BI Connector does the following:

- Provides the BI tool with the schema of the MongoDB collections to be visualized. Users can review the schema output to ensure data types, sub-documents, and arrays are correctly represented
- Translates SQL statements issued by the BI tool into equivalent MongoDB queries that are then sent to MongoDB for processing
- Converts the returned results into the tabular format expected by the BI tool, which can then visualize the data based on user requirements

MongoDB Charts

MongoDB Charts is the quickest and easiest way create and share visualisations of your MongoDB data in real time, without needing to move your data into other systems, or leverage third-party tools. Because Charts natively understands the MongoDB document model, you can create charts from data that varies in shape or contains nested documents and arrays, without needing to first map the data into a flat, tabular structure.

Using Charts, you can place multiple visualisations onto a single dashboard, and then share it with key stakeholders to support collaborative analysis and decision making, or you can embed them directly in your applications. When you connect to a live data source, MongoDB Charts will keep your visualizations and dashboards up to date with the most recent data. Charts will automatically generate an aggregation pipeline from your chart design, which is then executed on your MongoDB server. With MongoDB's workload isolation capabilities – enabling you to separate your operational from analytical workloads in the same cluster – you can use Charts for a real-time view **without having any impact on production workloads**.

You can get started by trying out MongoDB Charts.

Multi-Record ACID Transactional Model

Because documents can bring together related data that would otherwise be modelled across separate parent-child tables in a tabular schema, MongoDB's atomic single-document operations provide transaction semantics that meet the data integrity needs of the majority of applications. One or more fields may be written in a single operation, including updates to multiple sub-documents and elements of an array. The guarantees provided by MongoDB ensure complete isolation as a document is updated; any errors cause the operation to roll back so that clients receive a consistent view of the document.

MongoDB 4.0 added support for multi-document ACID transactions in the 4.0 release and extended them in 4.2 with Distributed Transactions that operate across scaled-out, sharded clusters. Multi-document 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 – multi-statement, 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 multi-document transactions, there are several best practices that developers should observe.

You can review all best practices in the MongoDB documentation for multi-document 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.

When reading data from any cluster member, users can tune MongoDB's consistency model to match application requirements, down to the level of individual queries within an app. When a situation mandates the strictest linearizable or causal consistency, MongoDB will enforce it; if an application needs to only read data that has been committed to a majority of nodes (and therefore can't be rolled back in the event of a primary election) or even just to a single replica, MongoDB can be configured for this. By providing this level of tunability, MongoDB can satisfy the full range of consistency, performance, and geo-locality requirements of modern apps.

Write Durability

MongoDB uses write concerns to control the level of write guarantees for data durability. Configurable options extend from simple 'fire and forget' operations to waiting for acknowledgments from multiple, globally distributed replicas.

If opting for the most relaxed write concern, the application can send a write operation to MongoDB then continue processing additional requests without waiting for a response from the database, giving the 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)

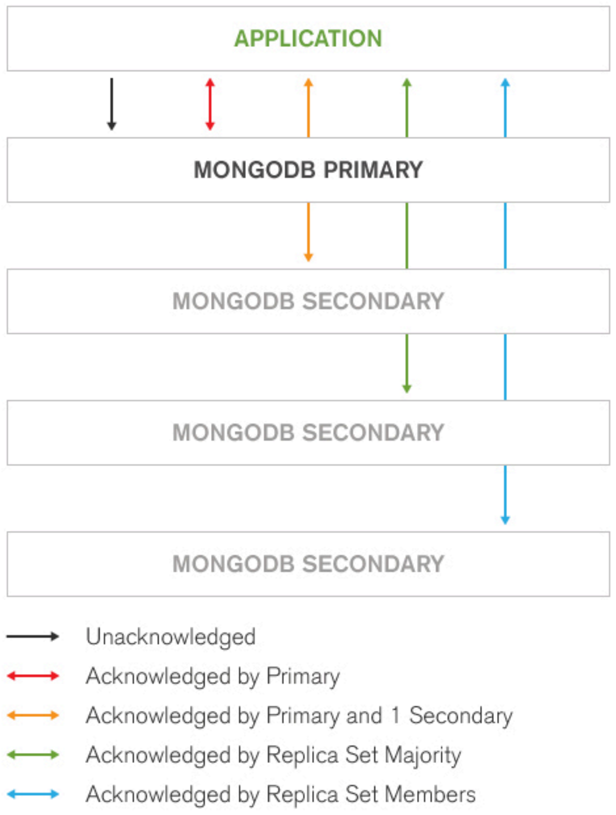


Figure 9: Configure Durability per Operation

The write concern can also be used to guarantee that the change has been persisted to disk before it is acknowledged.

The write concern is configured through the driver and is highly granular – it can be set per-operation, per-collection, or for the entire database. Users can learn more about write concerns in the documentation.

MongoDB uses write-ahead logging to an on-disk journal to guarantee write operation durability and to 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 & Constraints

Foreign Keys

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

With the document model, 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.

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 be either be rejected, or still written to the collection while logging a warning message. Structure can be imposed on just a subset of fields – for example requiring a valid customer a name and address, while others fields can be freeform, such as social media handle and cellphone 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.

As an example, 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), *price* (number no smaller than 0)
    - May optionally contain a boolean named *purchased*
    - Must contain no further fields

```
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"]
  }}
})
```

Enforcing Constraints With Indexes

As discussed in the Schema Design section, MongoDB supports unique indexes natively, which detect and raise an error to any insert operation that attempts to load a duplicate value into a collection. A tutorial is available that describes how to create unique indexes and eliminate duplicate entries from existing collections.

On-Demand Materialized Views

Materialized views are a common feature in relational databases, giving you the ability to pre-compute and store the results of common analytics queries. Typical use cases for them 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 now 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 MongoDB Charts, or the BI and Apache Spark connectors.

On-Demand Materialized Views represent a powerful addition to the analytics capabilities offered by the MongoDB database – enabling your 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.

Migrating Data to MongoDB

Project teams have multiple options for importing data from existing relational databases to MongoDB. The tool of choice should depend on the stage of the project and the existing environment.

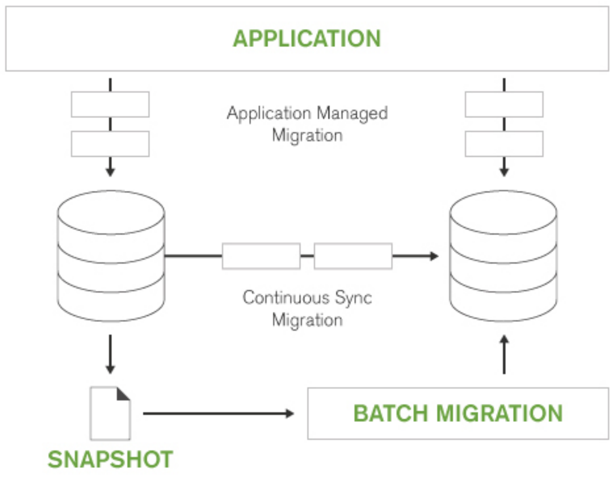


Figure 10: Multiple Options for Data Migration

Many users create their own scripts, which transform source data into a hierarchical JSON structure that can be imported into MongoDB using the `mongoimport` tool.

Extract Transform Load (ETL) tools are also commonly used when migrating data from relational databases to MongoDB. A number of ETL vendors including Informatica, Pentaho, and Talend have developed MongoDB connectors that enable a workflow in which data is extracted from the source database, transformed into the target MongoDB schema, staged, and then loaded into collections.

Many migrations involve running the existing RDBMS in parallel with the new MongoDB database, incrementally transferring production data:

- As records are retrieved from the RDBMS, the application writes them back out to MongoDB in the required document schema.
- Consistency checkers, for example using MD5 checksums, can be used to validate the migrated data.
- All newly created or updated data is written to MongoDB only.

Shutterfly used this incremental approach to migrate the metadata of 6 billion images and 20TB of data from Oracle to MongoDB.

Incremental migration can be used when new application features are implemented with MongoDB, or where multiple applications are running against the legacy RDBMS. Migrating only those applications that are being modernized enables teams to divide projects into more manageable and agile development sprints.

Incremental migration eliminates disruption to service availability while also providing fail-back should it be necessary to revert back to the legacy database.

Many organizations create feeds from their source systems, dumping daily updates from an existing RDBMS to MongoDB to run parallel operations, or to perform application development and load testing. When using this approach, it is important to consider how to handle deletes to data in the source system. One solution is to create “A” and “B” target databases in MongoDB, and then alternate daily feeds between them. In this scenario, Database A receives one daily feed, then the application switches the next day of feeds to Database B. Meanwhile the existing Database A is dropped, so when the next feeds are made to Database A, a whole new instance of the source database is created, ensuring synchronization of deletions to the source data.

Operational Agility at Scale: MongoDB Atlas

The considerations discussed thus far fall into the domain of the data architects, developers, and DBAs. However, no matter how elegant the data model or how efficient the indexes, none of this matters if the database fails to perform reliably at scale or cannot be managed efficiently.

An increasing number of companies are moving to the public cloud to not only reduce the operational overhead of managing infrastructure, but also 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.

MongoDB offers the fully managed, on-demand and global MongoDB Atlas service in the public cloud. Atlas enables you to take advantage of MongoDB’s capabilities on AWS, Azure, or GCP without needing to deploy, operate, and scale the software or underlying infrastructure yourself.

MongoDB Atlas is available through a pay-as-you-go model and billed on an hourly basis. It's easy to get started – use a simple GUI or programmatic API calls to select the public cloud provider, region, instance size, and features you need – all configured with operational best practices that leave you free to concentrate on your app, rather than backend database operations. MongoDB Atlas provides:

- Automated database and infrastructure provisioning along with auto-scaling so teams can get the database resources they need, when they need them, and can scale elastically in response to application demands.
- Security features to protect your data, with network isolation, fine-grained access control, auditing, and end-to-end encryption of data in-motion, in-use, and at-rest.
- Certifications against global standards to help you achieve your compliance requirements, including ISO 27001, SOC 2, and more. Atlas can be used for workloads subject to regulatory standards such as HIPAA, PCI-DSS, and GDPR.
- Multi-region Replication and Global Clusters to create geographically distributed deployments offering resilience, scale, and data residency compliance.
- 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 to your 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
- Access to the MongoDB Realm application platform, with QueryAnywhere, Functions, and Static Hosting delivered in a completely serverless model.
- The Full-Text Search service, providing rich search capabilities against your fully managed databases with no additional infrastructure to provision, manage, or scale.
- Live migration to move your self-managed MongoDB clusters into the Atlas service or to move Atlas clusters between cloud providers.

MongoDB Atlas is serving a vast 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 easily move from on-premises to the public cloud, and between providers as your needs evolve.

If you prefer to run MongoDB on infrastructure you manage, then check out our Production Notes along with the Operations Checklist.

Enabling your Teams: MongoDB University

Training courses are available for both developers and DBAs:

- **Free, web-based classes,** delivered over 7 weeks, supported by lectures, homework and forums to interact with instructors and other students. Over 1 million students have already enrolled in these classes.
- **Public training events** held at MongoDB facilities.
- **Private training** customized to an organization's specific requirements, delivered at their site.

Learn more.

Conclusion

Following the best practices outlined in this guide can help project teams reduce the time and risk of database migrations, while enabling them to take advantage of the benefits of MongoDB and the document model. In doing so, they 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 15,000 organizations rely on our commercial products. We offer software and services to make your life easier:

MongoDB Atlas is the database as a service for MongoDB, available on AWS, Azure, and GCP. It lets you focus on apps instead of ops. With MongoDB Atlas, you only pay for what you use with a convenient hourly billing model. Atlas auto-scales in response to application demand with no downtime, offering full security, resilience, and high performance.

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 visualizations of MongoDB data anywhere. 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.

MongoDB Stitch is a serverless platform which accelerates application development with simple, secure access to data and services from the client – getting your apps to market faster while reducing operational costs and effort.

MongoDB Realm will combine Realm, the popular mobile database and data synchronization technology, and MongoDB Stitch, the serverless platform for MongoDB, into a unified solution that makes it easy for you to build powerful and engaging experiences on more devices..

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 Stitch Serverless Platform (mongodb.com/cloud/stitch)
MongoDB Realm (mongodb.com/realm)