RDBMS to MongoDB Migration Guide

Considerations and Best Practices
June 2018

mongoDB

Table of Contents

Introduction

The relational database has been the foundation of enterprise data management for over thirty 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

Before considering technologies and architecture, a key to success is involving all key stakeholders for the application, including the line of business, developers, data architects, DBAs, and systems administrators. In some organizations, these roles may be combined.

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

The project team should work together to define business and technical objectives, timelines, and responsibilities, meeting regularly to monitor progress and address any issues.
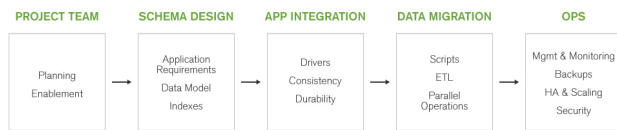
Figure 2: Migration Roadmap

There are a range of services and resources from MongoDB and the community that help build MongoDB skills and proficiency, including free, online training, support, and consulting. See the MongoDB Services section later in this guide for more detail.

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.

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

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

Figure 3: Terminology Translation

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.


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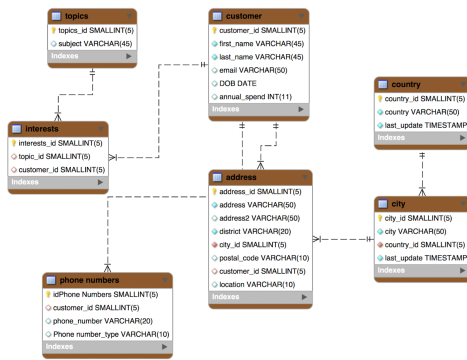
Figure 5: Modeling a customer with the relational database: data is split across multiple tables

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":
      ObjectId("5ad88534e3632e1a35a58d00"),
  "name": {
    "first": "John",
    "last": "Doe" },
  "address": [
    { "location": "work",
      "address": {
        "street": "16 Hatfields",
        "city": "London",
        "postal_code": "SE1 8DJ"},
      "geo": { "type": "Point", "coord": [
        51.5065752,-0.109081]}},
+   {...}
  ],
  "phone": [
    { "location": "work",
      "number": "+44-1234567890"},
+   {...}
  ],
  "dob": ISODate("1977-04-01T05:00:00Z"),
  "retirement_fund":
      NumberDecimal("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.

4

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 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. An example of this process is documented in the Apollo Group's migration from Oracle to MongoDB when

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

developing a new cloud-based learning management platform.

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.

Not all 1:1 and 1:m relationships should be embedded in a single document. Referencing between documents in different collections should be used when:

- A document is frequently read, but contains an embedded document that is rarely accessed. An example might be a customer record that embeds copies of the annual general report. Embedding the report 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

Referencing

Referencing enables data normalization, and can give more flexibility than embedding. But the application will issue follow-up queries to resolve the reference, requiring additional round-trips to the server, or require a JOIN operation using the $lookup aggregation stage.

References are usually implemented by saving the `_id` field[1] of one document in the related document as a reference. A second query is then executed by the application to return the referenced data.

Referencing should be used:

- With m:1 or m:m relationships where embedding would not provide sufficient read performance advantages to outweigh the implications of data duplication

- Where the object is referenced from many different sources

- To represent complex many-to-many relationships

The `$lookup` stage in an aggregation pipeline can be used to match the references with the `_ids` from the second collection to automatically embed the referenced data in the result set.

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 considerations, patterns and examples including embedded versus referenced relationships are discussed in more detail in the documentation.

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. Any field can be used for a secondary index, including fields within sub-documents and arrays.

Index options for MongoDB include:

---

1. A required unique field used as the primary key within a MongoDB document, either generated automatically by the driver or specified by the user.

- **Compound Indexes.** Using index intersection MongoDB can use more than one index to satisfy a query. This capability is useful when running ad-hoc queries as data access patterns are typically not known in advance. Where a query that accesses data based on multiple predicates is known, it will be more performant to use Compound Indexes, which use a single index structure to maintain 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.

- **Sparse Indexes.** Sparse indexes only contain entries for documents that contain the specified field. Because MongoDB allows the data model to vary from one document to another, it is common for some fields to be present only in a subset of all documents. Sparse indexes allow for smaller, more efficient indexes when fields are not present in all documents.

- **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.

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 RDBMSs.

Optimizing Performance With Indexes

MongoDB's query optimizer selects the index empirically by occasionally running alternate query plans and selecting

Figure 7: Visual Query Profiling in MongoDB Ops Manager

the plan with the best response time. The query optimizer can be overridden using the `cursor.hint()` method.

As with a relational database, the DBA can review query plans and ensure common queries are serviced by well-defined indexes by using the `explain()` function which reports on:

- The number of documents returned

- Which index was used – if any

- Whether the query was covered, meaning no documents needed to be read to return results

- Whether an in-memory sort was performed, which indicates an index would be beneficial

- The number of index entries scanned

- The number of documents read

- How long the query took to resolve, reported in milliseconds

- Alternate query plans that were assessed but then rejected

While it may not be necessary to shard 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 auto-sharding for application-transparent scale-out.

MongoDB provides a range of logging and monitoring tools to ensure collections are appropriately indexed and queries are tuned. These can and should be used both in development and in production.

The MongoDB Database Profiler is most commonly used during load testing and debugging, logging all database operations or only those events whose duration exceeds a configurable threshold (the default is 100ms). Profiling data is stored in a capped collection where it can easily be searched for relevant events – it is often easier to query this collection than parsing the log files.

Delivered as part of MongoDB's Ops Manager and Cloud Manager platforms, the Visual Query Profiler provides a quick and convenient way for operations teams and DBAs to analyze specific queries or query families. The Visual Query Profiler (as shown in Figure 7) displays how query and write latency varies over time – making it simple to identify slower queries with common access patterns and characteristics, as well as identify any latency spikes.

The visual query profiler will analyze data it collects to provide recommendations for new indexes that can be created to improve query performance. Once identified, these new indexes need to be rolled out in the production

system and Ops/Cloud Manager automates that process – performing a rolling index build which avoids any impact to the application.

MongoDB Compass provides the ability to visualize explain plans, presenting key information on how a query performed – for example the number of documents returned, execution time, index usage, and more. Each stage of the execution pipeline is represented as a node in a tree, making it simple to view explain plans from queries distributed across multiple nodes.

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 10+ developed and supported by MongoDB (e.g., Java, Python, .NET, PHP) 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 Ruby developers, PHP 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 to run MapReduce jobs 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`, subquery, 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 business users can seamlessly visualize semi-structured
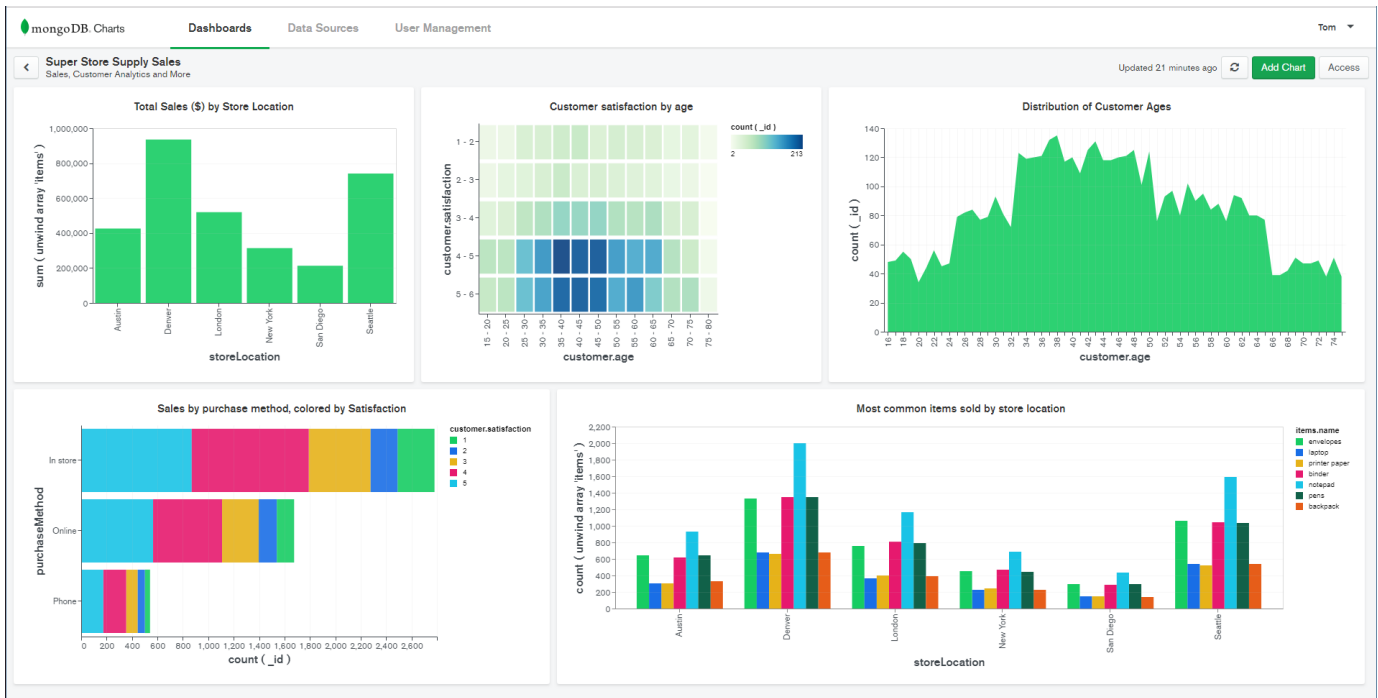
Figure 8: Creating rich graphs and dashboards directly from the Charts UI

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 a new tool that enables you to quickly and easily 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 quickly and easily visualize your data, place multiple charts onto a single dashboard, and then share that dashboard with key stakeholders to support collaborative analysis and decision making. 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**.

Initial chart types supported include bar, column, line, area, scatter, and donut charts. When you need more complex visualizations, or want to blend data from MongoDB with other data sources, the MongoDB Connector for BI provides integration with all leading SQL-based BI

platforms such as Tableau, Qlik, SAP Business Objects, and more.

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 adds support for multi-document ACID transactions, making it even easier for developers to address more 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.

Creating long running transactions, or attempting to perform an excessive number of operations in a single ACID transaction can result in high pressure on WiredTiger's cache. This is because the cache must maintain state for all subsequent writes since the oldest snapshot was created. As a transaction always uses the same snapshot while it is running, new writes accumulate in the cache throughout the duration of the transaction. These writes cannot be flushed until transactions currently running on old snapshots commit or abort, at which time the transactions release their locks and WiredTiger can evict the snapshot. To maintain predictable levels of database performance, developers should therefore consider the following:

1. By default, MongoDB will automatically abort any multi-document transaction that runs for more than 60 seconds. Note that if write volumes to the server are low, you have the flexibility to tune your transactions for a longer execution time. To address timeouts, the transaction should be broken into smaller parts that allow execution within the configured time limit. You should also ensure your query patterns are properly optimized with the appropriate index coverage to allow fast data access within the transaction.

2. There are no hard limits to the number of documents that can be read within a transaction. As a best practice, no more than 1,000 documents should be modified within a transaction. For operations that need to modify more than this limit, developers should break the transaction into separate parts that process documents in batches.

3. In MongoDB 4.0, a transaction is represented in a single oplog entry, therefore must be within the 16MB document size limit. While an update operation only stores the deltas of the update (i.e., what has changed), an insert will store the entire document. As a result, the combination of oplog descriptions for all statements in the transaction must be less than 16MB. If this limit is exceeded, the transaction will be aborted and fully rolled back. The transaction should therefore be decomposed into a smaller set of operations that can be represented in 16MB or less.

4. When a transaction aborts, an exception is returned to the driver and the transaction is fully rolled back. Developers should add application logic that can catch and retry a transaction that aborts due to temporary exceptions, such as a transient network failure or a primary replica election. With retryable writes, the MongoDB drivers will automatically retry the commit statement of the transaction.

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.
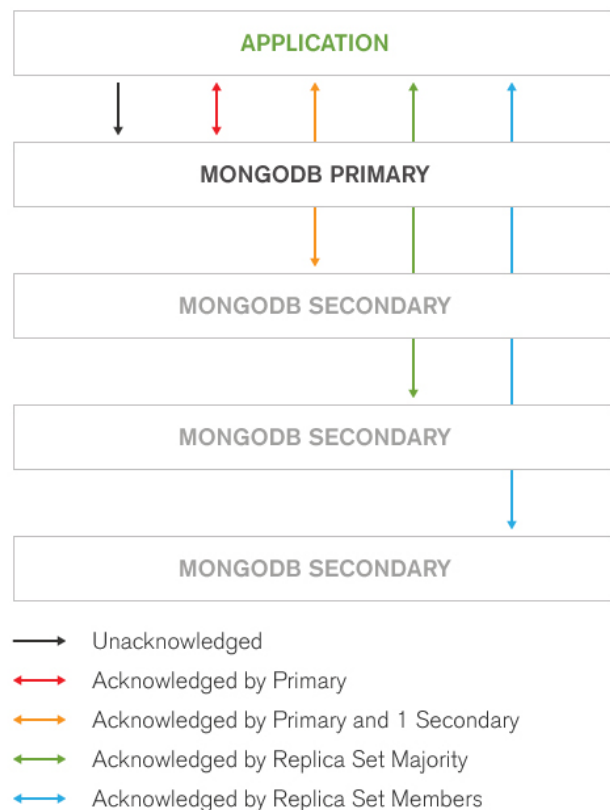
Write Durability

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 stronger write concerns, write operations wait until MongoDB applies and acknowledges the operation. This is MongoDB's default configuration. 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

As demonstrated earlier, MongoDB's document model can often eliminate the need for JOINs by collapsing parent/child tables and relationships into a single, rich BSON document. This same data model can also reduce the need for Foreign Key integrity constraints.

If you do want Foreign Key behavior from the database then third party frameworks such as Mongoose or PyMODM can be used.

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, DevOps and DBA teams 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.

Views

DBAs can define non-materialized views that expose only a subset of data from an underlying collection, i.e. a view that filters out specific fields. DBAs can define a view of a collection that's generated from an aggregation over another collection(s) or view. Permissions granted against

the view are specified separately from permissions granted to the underlying collection(s).

Views are defined using the standard MongoDB Query Language and aggregation pipeline. They allow the inclusion or exclusion of fields, masking of field values, filtering, schema transformation, grouping, sorting, limiting, and joining of data using `$lookup` and `$graphLookup` to another collection.

You can learn more about MongoDB read-only views from the documentation.

Developers or DBAs more familiar with relational schemas can view documents as conventional tables using MongoDB Compass.

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.

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.
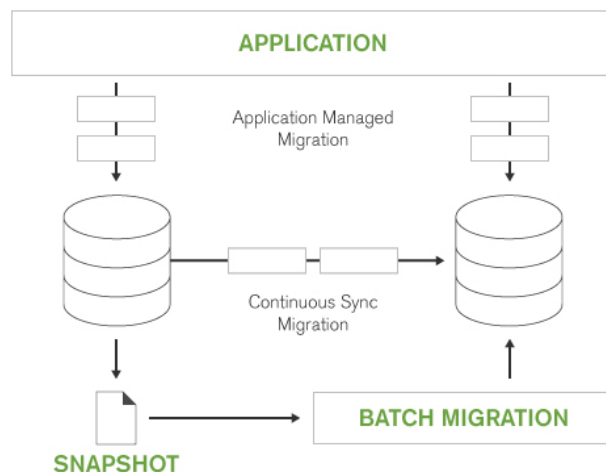
Figure 10: Multiple Options for Data Migration

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

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.

The final set of considerations in migration planning should focus on operational issues.

The MongoDB Operations Best Practices guide is the definitive reference to learn more on this key area.

The Operations Guide discusses:

- Management, monitoring, and backup with MongoDB Ops Manager or MongoDB Cloud Manager, which is the best way to run MongoDB within your own data center or public cloud, along with tools such as `mongotop`, `mongostat`, and `mongodump`

- High availability with MongoDB Replica Sets, providing self-healing recovery from failures and supporting scheduled maintenance with no downtime

- Scalability using MongoDB auto-sharding (partitioning) across clusters of commodity servers, with application transparency

- Hardware selection with optimal configurations for memory, disk and CPU

- Security including LDAP, Kerberos and x.509 authentication, field-level access controls, user-defined roles, auditing, encryption of data in-flight and at-rest, and defense-in-depth strategies to protect the database

MongoDB Atlas: Database as a Service For MongoDB

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 elastic MongoDB Atlas service, in the public cloud. Atlas enables customers to deploy, operate, and scale MongoDB databases on AWS, Azure, or GCP in just a few clicks or programmatic API calls. 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 to select the public cloud provider, region, instance size, and features you need. MongoDB Atlas provides:

- Automated database and infrastructure provisioning so teams can get the database resources they need, when they need them, and can elastically scale whenever they need to.

- Security features to protect your data, with network isolation, fine-grained access control, auditing, and end-to-end encryption, enabling you to comply with industry regulations such as HIPAA.

- Built in replication both within and across regions for always-on availability.

- Global clusters allows you to deploy a fully managed, globally distributed database that provides low latency, responsive reads and writes to users anywhere, with strong data placement controls for regulatory compliance.

- Fully managed, continuous and consistent 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.

- Automated patching and single-click upgrades for new major versions of the database, enabling you to take advantage of the latest and greatest MongoDB features.

- Live migration to move your self-managed MongoDB clusters into the Atlas service or to move Atlas clusters between cloud providers.

- Widespread coverage on the major cloud platforms with availability in over 50 cloud regions across Amazon Web

Services, Microsoft Azure, and Google Cloud Platform. MongoDB Atlas delivers a consistent experience across each of the cloud platforms, ensuring developers can deploy wherever they need to, without compromising critical functionality or risking lock-in.

MongoDB Atlas can be used for everything from a quick Proof of Concept, to dev/test/QA environments, to powering production applications. The user experience across MongoDB Atlas, Cloud Manager, and Ops Manager is consistent, ensuring that you easily move from on-premises to the public cloud, and between providers as your needs evolve.

Built and run by the same team that engineers the database, MongoDB Atlas is the best way to run MongoDB in the cloud. Learn more or deploy a free cluster now.

MongoDB Stitch

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

Stitch represents the next stage in the industry's migration to a more streamlined, managed infrastructure. Virtual Machines running in public clouds (notably AWS EC2) led the way, followed by hosted containers, and serverless offerings such as AWS Lambda and Google Cloud Functions. These still required backend developers to implement and manage access controls and REST APIs to provide access to microservices, public cloud services, and of course data. Frontend developers were held back by needing to work with APIs that weren't suited to rich data queries.

The Stitch serverless platform addresses these challenges by providing four services:

- **Stitch QueryAnywhere**. Brings MongoDB's rich query language safely to the edge. An intuitive SDK provides full access to your MongoDB database from mobile and IoT devices. Authentication and declarative or programmable access rules empower you to control precisely what data your users and devices can access.

- **Stitch Functions**. Stitch's HTTP service and webhooks let you create secure APIs or integrate with microservices and server-side logic. The same SDK that accesses your database, also connects you with popular cloud services, enriching your apps with a single method call. Your custom, hosted JavaScript functions bring everything together.

- **Stitch Triggers**. Real-time notifications let your application functions react in response to database changes, as they happen, without the need for wasteful, laggy polling.

- **Stitch Mobile Sync** (coming soon). Automatically synchronizes data between documents held locally in MongoDB Mobile and your backend database, helping resolve any conflicts – even after the mobile device has been offline.

Whether building a mobile, IoT, or web app from scratch, adding a new feature to an existing app, safely exposing your data to new users, or adding service integrations, Stitch can take the place of your application server and save you writing thousands of lines of boilerplate code.

Supporting Your Migration: MongoDB Services

MongoDB and the community offer a range of resources and services to support migrations by helping users build MongoDB skills and proficiency. MongoDB services include training, support, forums, and consulting. Refer to the "We Can Help" section below to learn more about support from development through to production.

MongoDB University

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 750,000 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.

Community Resources and Consulting

In addition to training, there is a range of other resources and services that project teams can leverage:

- **Technical resources** for the community are available through forums on Google Groups, StackOverflow and IRC

- **Consulting packages** include health checks, custom consulting and access to dedicated Technical Account Managers. More details are available from the MongoDB Consulting page.

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 MongoDB experts. Over 6,600 organizations rely on our commercial products. We offer software and services to make your life easier:

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

MongoDB Atlas is a database as a service for MongoDB, letting you focus on apps instead of ops. With MongoDB Atlas, you only pay for what you use with a convenient hourly billing model. With the click of a button, you can

scale up and down when you need to, with no downtime, full security, and high performance.

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 Mobile (Beta) MongoDB Mobile lets you store data where you need it, from IoT, iOS, and Android mobile devices to your backend – using a single database and query language.

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 Enterprise Download (mongodb.com/download)
MongoDB Atlas database as a service for MongoDB (mongodb.com/cloud)
MongoDB Stitch backend as a service (mongodb.com/cloud/stitch)

mongoDB