RDBMS to MongoDB Migration Guide

Considerations and Best Practices
June 2016

Table of Contents

Introduction 1

Organizing for Success 1

Schema Design 2
- From Rigid Tables to Flexible and Dynamic BSON Documents 3
  - Other Advantages of the Document Model 4
  - Joining Collections for Data Analytics 5
  - Defining the Document Schema 5
- Modeling Relationships with Embedding and Referencing 5
  - Embedding 5
  - Rerencing 6
  - Different Design Goals 6
- Indexing 7
  - Index Types 7
  - Optimizing Performance With Indexes 8
- Schema Evolution and the Impact on Schema Design 9

Application Integration 10
- MongoDB Drivers and the API 10
- Mapping SQL to MongoDB Syntax 10
- MongoDB Aggregation Framework 10
- MongoDB Connector for BI 11
- Atomicity in MongoDB 11
- Maintaining Strong Consistency 12
- Write Durability 12
- Implementing Validation & Constraints 13
  - Foreign Keys 13
  - Document Validation 13
  - Enforcing Constraints With Indexes 14

Migrating Data to MongoDB 14

Operational Agility at Scale 15

MongoDB Atlas: Database as a Service For MongoDB 15

Supporting Your Migration: MongoDB Services 16
- MongoDB University 16
- Community Resources and Consulting 16

Conclusion 16

We Can Help 16

Resources 17

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 growing 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 like MongoDB or NoSQL databases.

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.

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

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

Organization	Migrated From	Application
eHarmony	Oracle & Postgres	Customer Data Management & Analytics
Shutterfly	Oracle	Web and Mobile Services
Cisco	Multiple RDBMS	Analytics, Social Networking
Craigslist	MySQL	Archive
Under Armour	Microsoft SQL Server	eCommerce
Foursquare	PostgreSQL	Social, Mobile Networking Platforms
MTV Networks	Multiple RDBMS	Centralized Content Management
Buzzfeed	MySQL	Real-Time Analytics
Verizon	Oracle	Single View, Employee Systems
The Weather Channel	Oracle & MySQL	Mobile Networking Platforms

Figure 1: Case Studies

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 worlds.

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.

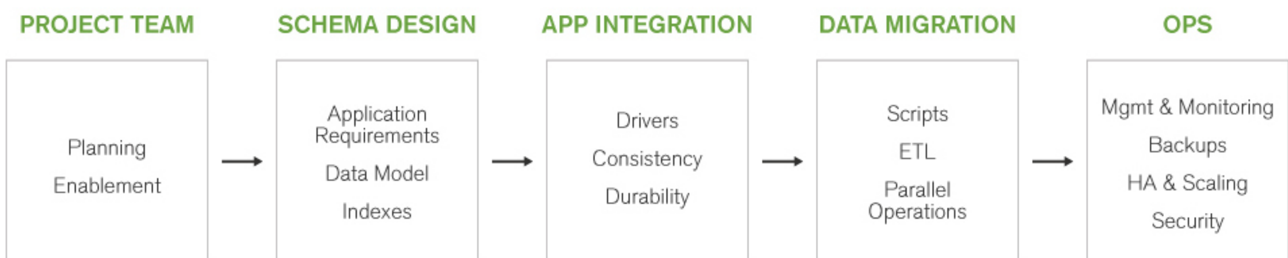

Figure 2: Migration Roadmap

RDBMS	MongoDB
Database	Database
Table	Collection
Row	Document
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, 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

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

commonly be collapsed (embedded) into a single document in MongoDB.

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. The document model enables data to be represented in a much more natural and intuitive way.

To further illustrate the differences between the relational and document models, consider the example of a blogging platform in Figure 5. In this example, the application relies on the RDBMS to join five separate tables in order to build the blog entry. With MongoDB, all of the blog data is contained within a single document, linked with a single reference to a user document that contains both blog and comment authors.

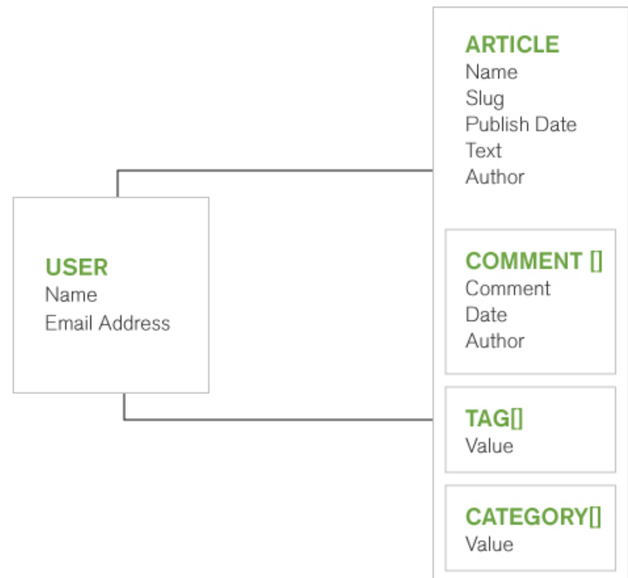

Figure 5: Pre-JOINing Data to Collapse 5 RDBMS Tables to 2 BSON Documents

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 massive 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 for Data Analytics

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 – MongoDB 3.2 adds that capability with 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.

While not offering as rich a set of join operations as some RDBMSs, `$lookup` provides a left outer equi-join which provides convenience for a selection of analytics use cases. A left outer equi-join, matches and embeds documents from the "right" collection in documents from the "left" collection.

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.

Worked examples of using `$lookup` as well as other agggregation stages can be found in the blog – Joins and Other Aggregation Enhancements.

Defining the Document Schema

The application's data access patterns should govern schema design, with specific understanding of:

- 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 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 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

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

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.

Architects should also embed fields that need to be modified together atomically. (Refer to the Application Integration section of this guide for more information.)

Not all 1:1 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 document size exceeds MongoDB's current 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.

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:

- When 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
- To model large, hierarchical data sets.

The `$lookup` stage in an aggregation pipeline can be used to match the references with the `_id`s 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)

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.

- MongoDB's document model is optimized for how the application accesses data (as developer time and speed to market are now more expensive than storage)

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 arrays.

Index options for MongoDB include:

- **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. By default, all indexes are not set as unique. 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 a field for 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's 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.** MongoDB 3.2 introduces Partial Indexes which 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.

MongoDB's storage engines all support all index types and the 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 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

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 new 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.

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.

Figure 7: Visual Query Profiling in MongoDB Ops Manager

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 contacts within each customer can be different
- The information stored on each of these contacts 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 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, DBA and operations teams to synchronize application and database upgrades, agreeing

on when to schedule the necessary `ALTER TABLE` operations.

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, Document 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 object-oriented programming, makes application integration simple.

MongoDB has idiomatic drivers for the most popular languages, including eleven 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 Framework

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 Framework natively within the database, which delivers similar functionality to the `GROUP BY` and related SQL statements.

When using the Aggregation Framework, documents in a collection pass through an aggregation pipeline, 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,

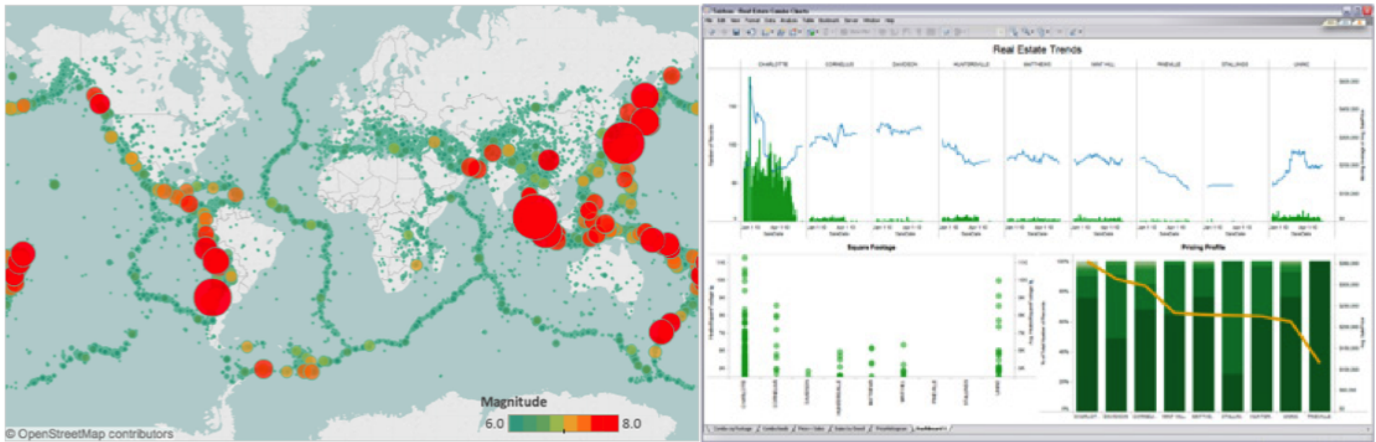

Figure 8: Uncover new insights with powerful visualizations generated from MongoDB

minimums, averages, standard deviations and related data) as documents progress through the pipeline.

The SQL to Aggregation Mapping Chart shows a number of examples demonstrating how queries in SQL are handled in MongoDB's Aggregation Framework. To enable more complex analysis, MongoDB also provides native support for MapReduce operations in both sharded and unsharded collections.

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 for the first time be easily explored with industry-standard SQL-based BI and analytics platforms. Using the BI Connector, analysts, data scientists and business users can now 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 collection 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

Additionally, a number of Business Intelligence (BI) vendors have developed connectors to integrate MongoDB with their suites (without using SQL), alongside traditional relational databases. This integration provides reporting, ad hoc analysis, and dashboarding, enabling visualization and analysis across multiple data sources. Integrations are available with tools from a range of vendors including Actuate, Alteryx, Informatica, JasperSoft, Logi Analytics, MicroStrategy, Pentaho, QlikTech, SAP Lumira and Talend.

Atomicity in MongoDB

Relational databases typically have well developed features for data integrity, including ACID transactions and constraint enforcement. Rightly, users do not want to sacrifice data integrity as they move to new types of databases. With MongoDB, users can maintain many capabilities of relational databases, even though the

technical implementation of those capabilities may be different.

MongoDB write operations are ACID at the document level – including the ability to update embedded arrays and sub-documents atomically. By embedding related fields within a single document, users get the same integrity guarantees as a traditional RDBMS, which has to synchronize costly ACID operations and maintain referential integrity across separate tables.

Document-level atomicity in MongoDB ensures complete isolation as a document is updated; any errors cause the operation to roll back and clients receive a consistent view of the document.

Despite the power of single-document atomic operations, there may be cases that require multi-document transactions. There are multiple approaches to this – including using the `findAndModify` command that allows a document to be updated atomically and returned in the same round trip. `findAndModify` is a powerful primitive on top of which users can build other more complex transaction protocols. For example, users frequently build atomic soft-state locks, job queues, counters and state machines that can help coordinate more complex behaviors. Another alternative entails implementing a two-phase commit to provide transaction-like semantics. The documentation describes how to do this in MongoDB, and important considerations for its use.

Maintaining Strong Consistency

By default, MongoDB directs all read operations to primary servers, ensuring strong consistency. Also, by default any reads from secondary servers within a MongoDB replica set will be eventually consistent – much like master / slave replication in relational databases.

Administrators can configure the secondary replicas to handle read traffic using MongoDB's Read Preferences, which control how clients' read operations are routed to members of a replica set.

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.

With a 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 acknowledges the operation. This is MongoDB's default configuration. MongoDB provides multiple levels of write concern to address specific application requirements. For example:

- The application waits for acknowledgment from the primary server (default).
- Or the write operation is also replicated to one secondary.
- Or the write operation is also replicated to a majority of secondaries.
- Or write operation is also replicated to 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).

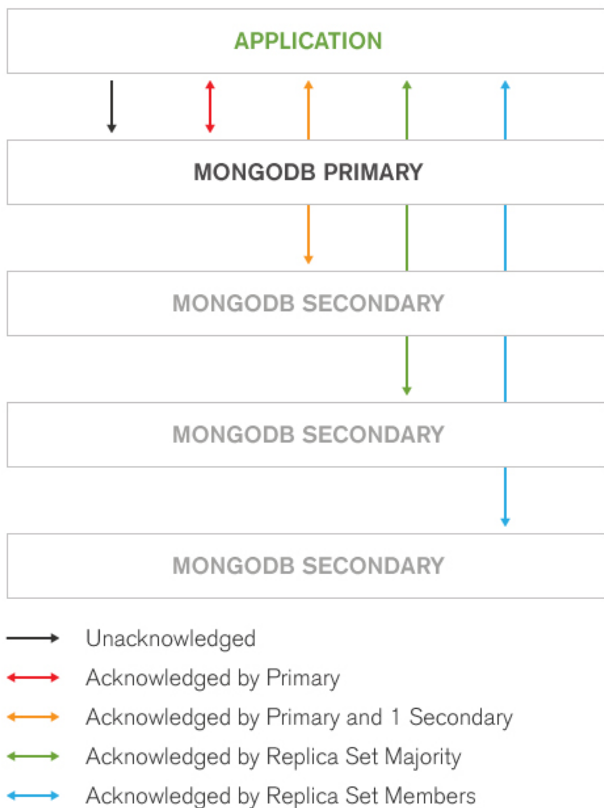

Figure 9: Configure Durability per Operation

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 in 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 embedding data within a single, atomically updated BSON document. This same data model can also reduce the need for Foreign Key integrity constraints.

Document Validation

Dynamic schemas bring great agility, but it is also important that controls can be implemented to maintain data quality, especially if the database is powering multiple applications, or is integrated into a larger data management platform that feeds into upstream and downstream systems. Rather than delegating enforcement of these controls back into application code, MongoDB provides Document Validation within the database. Users can enforce checks on document structure, data types, data ranges, and the presence of mandatory fields. As a result, DBAs can apply data governance standards, while developers maintain the benefits of a flexible document model.

There is significant flexibility to customize which parts of the documents are **and are not** validated for any collection – unlike an RDBMS where everything must be defined and enforced. For any key it might be appropriate to check:

- That it exists
- If it does exist, that the value is of the correct type
- That the value is in a particular format (regular expressions can be used to check if the contents of the string matches a particular pattern – that it's a properly formatted email address, for example)
- That the value falls within a given range

As an example, assume that the following checks need to be enforced on the `contacts` collection:

- The year of birth is no later than 1994
- The document contains a phone number and/or an email address
- When present, the phone number and email addresses are strings

That can be achieved by defining this Document Validation rule:

```
db.runCommand({
    collMod: "contacts",
    validator: {
        $and: [
            {year_of_birth: {$lte: 1994}},
            {$or: [
                {phone: { $type: "string" }},
                {email: { $type: "string" }}
    ]}]
}})
```

Adding the validation checks to a collection is very intuitive to any developer or DBA familiar with MongoDB, as Document Validation uses the standard MongoDB Query Language.

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.

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 then loaded into document collections.

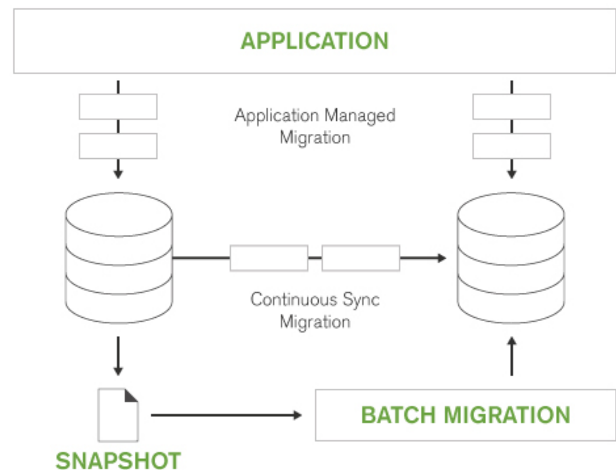

Figure 10: Multiple Options for Data Migration

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.

To learn more about each of these and other options including using Apache Hadoop for ETL, review the Migration Patterns webinar.

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

MongoDB Atlas provides all of the features of MongoDB, without the operational heavy lifting required for any new application. MongoDB Atlas is available on-demand through a pay-as-you-go model and billed on an hourly basis, letting you focus on what you do best.

It's easy to get started – use a simple GUI to select the instance size, region, and features you need. MongoDB Atlas provides:

- Security features to protect access to your data
- Built in replication for always-on availability, tolerating complete data center failure
- Backups and point in time recovery to protect against data corruption
- Fine-grained monitoring to let you know when to scale. Additional instances can be provisioned with the push of a button
- Automated patching and one-click upgrades for new major versions of the database, enabling you to take advantage of the latest and greatest MongoDB features
- A choice of cloud providers, regions, and billing options

MongoDB Atlas is versatile. It's great for everything from a quick Proof of Concept, to test/QA environments, to complete production clusters. If you decide you want to bring operations back under your control, it is easy to move your databases onto your own infrastructure and manage them using MongoDB Ops Manager or MongoDB Cloud Manager. The user experience across MongoDB Atlas, Cloud Manager, and Ops Manager is consistent, ensuring that disruption is minimal if you decide to migrate to your own infrastructure.

MongoDB Atlas is an ideal fit for organizations looking to MongoDB as a potential alternative to their relational database, and who don't want to tie up their operations teams for the evaluation process.

MongoDB Atlas is automated, it's easy, and it's from the creators of MongoDB. Learn more and take it for a spin.

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 350,000 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.

Learn more.

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 2,000 organizations rely on our commercial products, including startups and more than a third of the Fortune 100. 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 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 Professional helps you manage your deployment and keep it running smoothly. It includes support from MongoDB engineers, as well as access to MongoDB Cloud Manager.

Development Support helps you get up and running quickly. It gives you a complete package of software and services for the early stages of your project.

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.

Contact us to learn more, or visit mongodb.com.

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)

New York • Palo Alto • Washington, D.C. • London • Dublin • Barcelona • Sydney • Tel Aviv
US 866-237-8815 • INTL +1-650-440-4474 • info@mongodb.com
© 2016 MongoDB, Inc. All rights reserved.