MongoDB Architecture Guide

MongoDB 4.0
June 2018

Table of Contents

Introduction

The success of every business rests on its ability to use technology, and in particular software and data, to create a competitive advantage. Companies want to quickly develop new digital products and services to drive expansion of revenue streams, improve customer experience by engaging them in more meaningful ways, and identify opportunities to reduce the risk and cost of doing business.

Organizations turn to various strategies to enable these technology transformations:

- Aligning behind new IT models and processes, such as Agile and DevOps methodologies.
- Adopting new architectures and platforms by taking a mobile-first approach, moving to microservices patterns, and shifting underlying infrastructure to the cloud.
- Exploiting emerging technologies including AI and machine learning, IoT, and blockchain.

Despite these new strategies for tackling IT initiatives, transformation continues to be complex and slow. Research from a 2017 survey by Harvey Nash and KPMG[1] revealed that 88% of CIOs believe they have yet to benefit from their digital strategy.

Why is this the case? Data is at the heart of every application, and from our experience in working with organizations ranging from startups to many Fortune 100 companies, realizing its full potential is still a significant challenge:

- Demands for higher developer productivity and faster time to market – with release cycles compressed to days and weeks – are being held back by traditional rigid relational data models and waterfall development.
- The inability to manage massive increases in new, rapidly changing data types – structured, semi-structured, and polymorphic data generated by modern web, mobile, social, and IoT applications.
- Difficulty in exploiting the wholesale shift to distributed systems and cloud computing that enable developers to access on-demand, highly scalable compute and storage infrastructure, while meeting a whole new set of regulatory demands for data sovereignty.

1. https://home.kpmg.com/xx/en/home/insights/2017/05/harvey-nash-kpmg-cio-survey-2017.html

MongoDB responded to these challenges by creating a technology foundation that enables development teams through:

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

With these capabilities, you can build an Intelligent Operational Data Platform, underpinned by MongoDB. In this Guide, we dive deeper into the architecture of MongoDB that enables the three technology foundations described above.

The Best Way to Work with Data: The Document Model

Relational, tabular databases have a long-standing position in most organizations. This made them the default way to think about storing, using, and enriching data. But enterprises are increasingly encountering limitations of this technology. Modern applications present new challenges that stretch the limits of what's possible with a relational database.

As organizations seek to build these modern applications, they find that the key differentiator for success is their development teams. Developers are on the front lines of digital transformation, and enabling them to work faster produces compounding benefits for the organization. To realize the full potential of data and software, developers turn to technologies that enable rather than hinder them.

Through strategies such as Agile and DevOps, microservices, cloud replatforming and more, many organizations have made significant progress in refactoring and evolving application tier code to respond faster to changing business requirements. But they then find themselves hampered by the rigidity and complexity of tabular databases.

Organizations need a fresh way to work with data. In order to handle the complex data of modern applications and simultaneously increase development velocity, the key is a platform that is:

- **Easy**, letting them work with data in a natural, intuitive way, supported by strong data consistency and transactional data guarantees
- **Flexible**, so that they can adapt and make changes quickly
- **Fast**, delivering great performance with less code
- **Versatile**, supporting a wide variety of data models, relationships, and queries

MongoDB's document model delivers these benefits for developers, making it the best way to work with data.

Easy: Natural, Intuitive, Transactional

Relational databases use a tabular data model, storing data across many tables. An application of any complexity easily requires hundreds or even thousands of tables. This sprawl occurs because of the way the tabular model treats data.

The conceptual model of application code typically resembles the real world. That is, objects in application code, including their associated data, usually correspond to real-world entities: customers or users, products, IoT connected assets, and so on. Relational databases, however, require a different structure. Because of the need to normalize data, a logical entity is typically represented in many separate parent-child tables linked by foreign keys. This data model doesn't resemble the entity in the real world, or how that entity is expressed as an object in application code.

This difference makes it difficult for developers to reason about the underlying data model while writing code, slowing down application development; this is sometimes referred to as *object-relational impedance mismatch*. One workaround for this is to employ an object-relational mapping layer (ORM). But this creates its own challenges, including managing the middleware and revising the mapping whenever either the application code or the database schema changes.

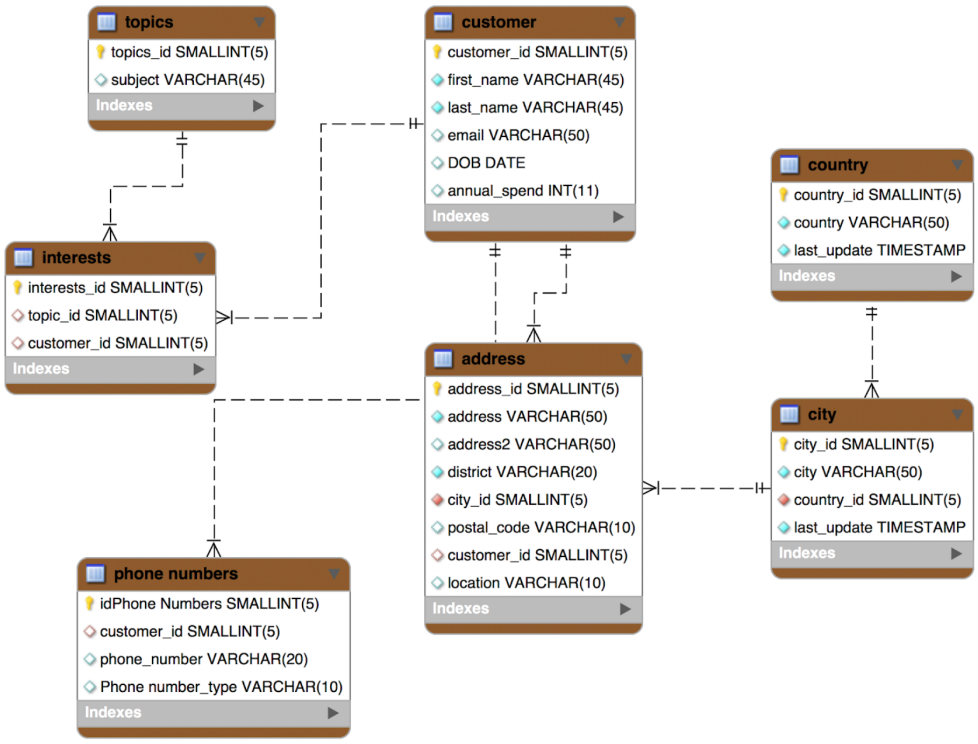

Figure 1: Modeling a customer with the relational database: data is split across multiple tables

In contrast to this tabular model, MongoDB uses a **document** data model. Documents are a much more natural way to describe data. They present a single data structure, with related data embedded as sub-documents and arrays. 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. It also significantly reduces the barrier-to-entry for new developers who begin working on a project – for example, adding new microservices to an existing app.

This JSON document demonstrates how a customer object is modeled in a single, rich document structure with nested arrays and sub-documents to collapse what are otherwise separate parent-child tables in a relational database.

```
{
  "_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")
}
```

MongoDB stores data as JSON (JavaScript Object Notation) documents in a binary representation called BSON (Binary JSON). Unlike other databases that store JSON data as simple strings and numbers, the BSON encoding extends the JSON representation to include

additional types such as int, long, date, floating point, and decimal128 – the latter is especially important for high precision, lossless financial and scientific calculations. This makes it much easier for applications to reliably process, sort, and compare data. BSON documents contain one or more fields, and each field contains a value of a specific data type, including arrays, binary data, and sub-documents.

MongoDB provides native drivers for all popular programming languages and frameworks to make development easy and natural. Supported drivers include Java, Javascript, C#/.NET, Python, Perl, PHP, Scala and others, in addition to 30+ community-developed drivers. MongoDB drivers are designed to be idiomatic for the given programming language.

MongoDB Compass, the GUI for MongoDB, makes it easy to explore and manipulate your data. Visualize the structure of data in MongoDB; visually create queries and aggregation pipelines from the GUI, evaluate their performance, and then export them as code to your app; view and create indexes; build data validation rules; and more. Compass provides an intuitive interface for working with MongoDB.

Data Consistency Guarantees

MongoDB's ease of use extends to data consistency requirements. As a distributed system, MongoDB handles the complexity of maintaining multiple copies of data via replication (see the Availability section later). 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.

When writing data, MongoDB similarly offers tunable configurations for durability requirements, discussed further in the Availability section.

ACID Transactional Model

Because documents can bring together related data that would otherwise be modeled 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 a complete range of use cases with MongoDB. They feel just like the transactions developers are familiar with from relational databases – multi-statement, similar syntax, and easy to add to any application. Through snapshot isolation, transactions provide a consistent view of data, enforce all-or-nothing execution, and do not impact performance for workloads that do not require them. Learn more by visiting the MongoDB multi-document transactions page where you can see code samples, documentation, and chalk and talks from the engineers who implemented them.

Flexibility: Dynamically Adapting to Changes

The tabular data model is rigid. It was built for structured data, where each record in a table has identical columns. While it's possible to handle polymorphism and semi-structured or unstructured data, it's clumsy, and working around the basic data limitations of the tabular model takes up development time. Furthermore, the tabular model demands that the schema be pre-defined, with any changes requiring schema migrations. Practically, this

means that developers need to plan their data model well in advance, and imposes friction to the development process when adding features or making application updates that require schema changes. This is a poor match for today's agile, iterative development models.

MongoDB documents are polymorphic – fields can vary from document to document within a single collection (analogous to table in a tabular database). For example, all documents that describe customers might contain the customer ID and the last date they purchased a product or service, but only some of these documents might contain the user's social media handle, or location data from a mobile app. There is no need to declare the structure of documents to the system – documents are self-describing. If a new field needs to be added to a document, the field can be created without affecting all other documents in the system, without updating a central system catalog, and without taking the database offline.

Developers can start writing code and persist objects as they are created. And when they need to add more features, MongoDB continues to store the updated objects without the need to perform costly `ALTER TABLE` operations – or worse, having to redesign the schema from scratch. 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 application code changed accordingly.

Schema Governance

While MongoDB's flexible schema is a powerful feature, there are situations where strict guarantees on the schema's data structure and content are required. Unlike NoSQL databases that push enforcement of these controls back to the developer to implement in application code, MongoDB provides schema validation within the database via syntax derived from the proposed IETF JSON Schema standard.

Using schema validation, DevOps and DBA teams can define a prescribed document structure for each collection, with the database rejecting any documents that do not conform to it. 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 written to the collection while logging a warning message. Structure can be imposed on just a subset of a document's fields – for example, requiring a valid customer name and address, while other fields can be freeform.

With schema validation, DBAs can apply data governance standards to their schema, while developers maintain the benefits of a flexible document model.

Speed: Great Performance

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

In MongoDB, a document is a single place for the database to read and write data for an entity. This locality of data ensures the complete document can be accessed in a single database operation that avoids the need internally to pull data from many different tables and rows. For most queries, there's no need to JOIN multiple records. Should your application access patterns require it, MongoDB does provide the equivalent of a JOIN, the ability to `$lookup`[2] between multiple collections. This is very useful for analytics workloads, but is generally not required for operational use cases.

The document model approach also simplifies query development and optimization. There's no need to write complex code to manipulate text and values into SQL and work with multiple tables. Figure 2 illustrates the difference between using the MongoDB query language[3] and SQL[4] to insert a single user record, where users have multiple properties including name, all of their addresses, phone numbers, interests, and more.

2. https://docs.mongodb.com/manual/reference/operator/aggregation/lookup/index.html
3. https://git.io/vpnxX
4. https://git.io/vpnpG

SQL

```
def addUser(connection, user):
    cursor = connection.cursor()

    customerInsert = (
        "INSERT INTO customer (first_name, last_name,
            email, "
        "DOB, annual_spend) VALUES "
        "(%(first)s, %(last)s, %(email)s, %(dob)s, %(spend)
            s)")

    customerData = {
        'first': user['name']['first'],
        'last': user['name']['second'],
        'email': user['email'],
        'dob': user['dob'],
        'spend': user['annualSpend']
    }

    cursor.execute(customerInsert, customerData)
    customerId = cursor.lastrowid

    cityQuery = ("SELECT city_id FROM city WHERE city = %(
        city)s")
    for address in user['address']:
        cursor.execute(cityQuery, {'city': address['city']}
            )
        city_id = cursor.fetchone()[0]

        addressInsert = (
            "INSERT INTO address (address, address2,
                district, "
            "city_id, postal_code, customer_id, location) "
            "VALUES (%(add)s, %(add2)s, %(dist)s, %(city)s
                , %(post)s, %(cust)s, %(loc)s)")

        addressData = {
            'add': address['number'],
            'add2': address['street'],
            'dist': address['state'],
            'city': city_id,
            'post': address['postalCode'],
            'cust': customerId,
            'loc':  address['location']
        }
        cursor.execute(addressInsert, addressData)

    picQuery = ("SELECT topics_id FROM topics WHERE
      subject = %(subj)s")
    terestInsert = (
      "INSERT into interests (topic_id, customer_id) "
      "VALUES (%(topic)s, %(cust)s)")

    r interest in user['interests']:
      topicId = 0
      topicData = {
          'subj': interest['interest']
      }

      cursor.execute(topicQuery, topicData)
      row = cursor.fetchone()
      if row is None:
          topicInsert = ("INSERT INTO topics (subject)
              VALUES (%(subj)s)")
          cursor.execute(topicInsert, topicData)
          topicId = cursor.lastrowid
      else:
          topicId = row[0]

      interestData = {
          'topic': topicId,
          'cust': customerId
      }
      cursor.execute(interestInsert, interestData)

    oneInsert = (
      "INSERT INTO `phone numbers` (customer_id,
          phone_number, `Phone number_type`) "
      "VALUES (%(cust)s, %(num)s, %(type)s)")
    r phoneNumber in user['phone']:
      phoneData = {
          'cust': customerId,
          'num': phoneNumber['number'],
          'type': phoneNumber['location']
      }
      cursor.execute(phoneInsert, phoneData)

    nnection.commit()
    rsor.close()
  return customerId
```

MongoDB

```
def addUser(database, user):
    return database.customers.insert_one(user).inserted_id
```

Figure 2: Comparison of SQL and MongoDB code to insert a single user

Creating Real-Time Data Pipelines with Change Streams

Further building on the "speed" theme change streams enable developers to build reactive and real-time apps for web, mobile, and IoT that can view, filter, and act on data changes as they occur in the database. Change streams enable fast and seamless data movement across distributed database and application estates, making it simple to stream data changes and trigger actions wherever they are needed, using a fully reactive programming style. Use cases enabled by MongoDB change streams include:

- Powering trading applications that need to be updated in real time as stock prices rise and fall.
- Refreshing scoreboards in multiplayer games.
- Updating dashboards, analytics systems, and search engines as operational data changes.
- Creating powerful IoT data pipelines that can react whenever the state of physical objects change.
- Synchronizing updates across serverless and microservice architectures by triggering an API call when a document is inserted or modified.

Versatility: Any Data Model, Any Access Pattern

Building upon the ease, flexibility, and speed of the document model, MongoDB enables developers to satisfy a range of application requirements, both in the way data is modeled and how it is queried.

The flexibility and rich data types of documents make it possible to model data in many different structures, representative of entities in the real world. The embedding of arrays and sub-documents makes documents very powerful at modeling complex relationships and hierarchical data, with the ability to manipulate deeply nested data without the need to rewrite the entire document. But documents can also do much more: they can be used to model flat, table-like structures, simple key-value pairs, text, geospatial data, the nodes and edges used in graph processing, and more.

With an expressive query language documents can be queried in many ways (see Table 1) – from simple lookups and range queries to creating sophisticated processing pipelines for data analytics and transformations, through to faceted search, JOINs, geospatial processing, and graph traversals. This is in contrast to most distributed databases, which offer little more than simple key-value access to your data.

The MongoDB query model is also implemented as methods or functions within the API of a specific programming language, as opposed to a completely separate language like SQL. This, coupled with the affinity between MongoDB's JSON document model and the data structures used in object-oriented programming, further speeds developer productivity. For a complete list of drivers see the MongoDB Drivers documentation.

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

Table 1: MongoDB's rich query functionality

MongoDB's versatility is further supported by its indexing capabilities. Queries can be performed quickly and efficiently with an appropriate indexing strategy. MongoDB permits secondary indexes to be declared on any field, including field within arrays. Indexes can be created and dropped at any time to easily support changing application requirements and query patterns. Index types include compound indexes, text indexes, geospatial indexes, and more. Further, indexes can be created with special properties to enforce data rules or support certain workloads – for example, to expire data according to retention policies or guarantee uniqueness of the indexed field within a collection. Table 2 summarizes the indexes available with MongoDB.

Index Types	Index Features
Primary Index: Every Collection has a primary key index	**TTL Indexes**: Single Field indexes, when expired delete the document
Compound Index: Index against multiple keys in the document	**Unique Indexes**: Ensures value is not duplicated
MultiKey Index: Index into arrays	**Partial Indexes**: Expression based indexes, allowing indexes on subsets of data
Text Indexes: Support for text searches	**Case Insensitive Indexes**: supports text search using case insensitive search
GeoSpatial Indexes: 2d & 2dSphere indexes for spatial geometries	**Sparse Indexes**: Only index documents which have the given field
Hashed Indexes: Hashed based values for sharding	

Table 2: MongoDB offers fully-featured secondary indexes

MongoDB Stitch: Serverless Platform

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.

Put Data Where you Need It: Intelligent Distributed Systems Architecture

Mobile, web, IoT, and cloud apps have significantly changed user expectations. Once, applications were designed to serve a finite audience – typically internal business departments – in a single head office location. Now, users demand modern app experiences that must be always-on, accessible from any device, consistently scaled with the same low-latency responsiveness wherever they are while meeting the data sovereignty requirements demanded by new data privacy regulations.

To address these needs, MongoDB is built around an intelligent distributed systems architecture that enables developers to place data where their apps and users need it. MongoDB can be run within and across geographically distributed data centers and cloud regions, providing levels of availability, workload isolation, scalability, and data locality unmatched by relational databases. Before diving further into MongoDB's distributed systems design, let's first examine the challenges of meeting modern app needs with traditional relational databases.

Relational Database Challenges

Relational databases are monolithic systems, designed to run on a single server, typically with shared storage. Attempting to introduce distributed system properties to relational databases results in significantly higher developer and operations complexity and cost, slowing the pace of delivering new apps, and evolving them in line with user requirements, while struggling to meet the demands for always-on availability and scaling as the app grows.

Availability

For redundancy, most relational databases support replication to mirror the database across multiple nodes,

but they lack the integrated mechanisms for automatic failover and recovery between database replicas. As a result, users need to layer 3rd-party clustering frameworks and agents (sometimes called "brokers") to monitor the database and its host platform, initiating failover in the event something goes wrong (i.e., the database crashes or the underlying server stops responding). What are the downsides of this approach?:

- Failover events need to be coordinated by the clustering software across the database, replication mechanism, storage, network, clients, and hosts. As a result, it can take multiple minutes to recover service to the application, during which time, the app is unavailable to users.
- Clustering frameworks are often external to the database, so developers face the complexity of integrating and managing separate pieces of technology and processes, sometimes backed by different vendors. In some cases, these clustering frameworks are independently licensed from the database itself, adding cost.
- It also means additional complexity in coordinating the implementation, testing, and ongoing database maintenance across multiple teams – developers, DBAs, network administrators, and system administrators – each with their own specific areas of responsibility.

Scale-Out and Data Locality

Attempting to accommodate increasing data volumes and user populations with a database running on a single server means developers can rapidly hit a scalability wall, necessitating significant application redesign and custom engineering work. While it can be possible to use replication to scale read operations across replicas of the data – with potential risks to data consistency – relational databases have no native mechanisms to partition (shard) the database across a cluster of nodes when they need to scale writes. So developers are confronted with two options:

1. Manually partition the database at the application level, which adds significant development complexity, and inhibits the ability to elastically expand and contract the cluster as workloads dictate, or as the app scales beyond the original capacity predictions.
2. Integrate a separate sharding framework for the database. Like the HA frameworks discussed above, these sharding layers are developed independently from the database, so the user has the added complexity of integrating and managing multiple, distinct pieces of technology in order to provide a complete solution.

Whatever approach is taken to scaling a tabular database, developers will typically lose key relational capabilities that are at the heart of traditional RDBMS application logic: ACID transactions, referential integrity, JOINs, and full SQL expressivity for any operations that span shards. As a result, they will need to recreate this functionality back at the application tier.

MongoDB Distributed Systems Architecture

As a distributed data platform, MongoDB gives developers four essential capabilities in meeting modern application needs:

- Availability
- Workload isolation
- Scalability
- Data locality

Each is discussed in turn below.

Availability

MongoDB maintains multiple copies of data using replica sets (Figure 3). Unlike relational databases, replica sets are self-healing as failover and recovery is fully automated, so it is not necessary to manually intervene to restore a system in the event of a failure, or to add additional clustering frameworks and agents. Replica sets also provide operational flexibility by providing a way to perform systems maintenance (i.e. upgrading underlying hardware and software) using rolling replica restarts that preserve service continuity.

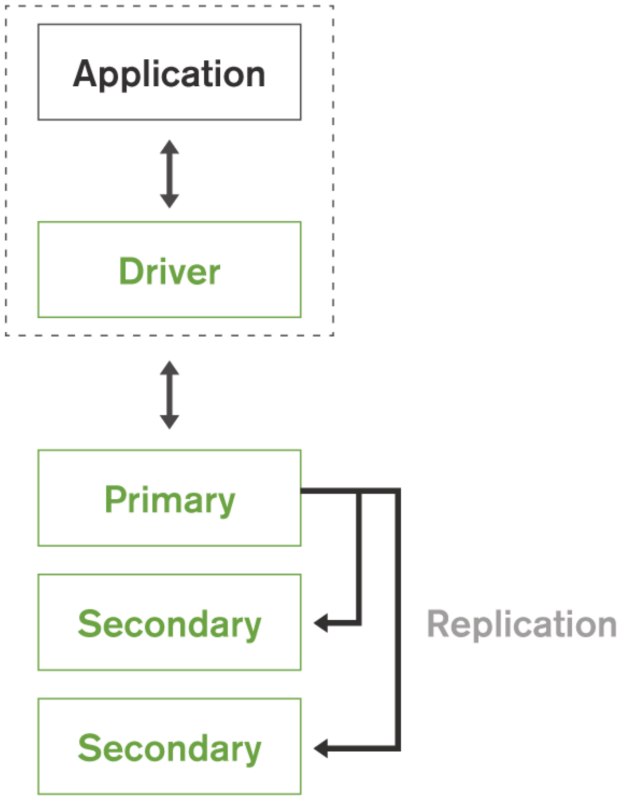

Figure 3: Self-healing MongoDB replica sets for continuous availability

A replica set consists of multiple database replicas. To maintain strong data consistency, one member assumes the role of the primary replica against which all write operations are applied (as discussed later, MongoDB automatically shards the data set across multiple nodes to scale write operations beyond a single primary node). The other members of the replica set act as secondaries, replicating all data changes from the oplog (operations log). The oplog contains an ordered set of idempotent operations that are replayed on the secondaries.

If the primary replica set member suffers a failure (e.g., a power outage, hardware fault, network partition), one of the secondary members is automatically elected to primary, typically within several seconds, and the client connections automatically failover to that new primary. Any writes that could not be serviced during the election can be automatically retried by the drivers once a new primary is established, with the MongoDB server enforcing exactly-once processing semantics. Retryable writes enable MongoDB to ensure write availability, without sacrificing data consistency.

The replica set election process is controlled by sophisticated algorithms based on an extended implementation of the Raft consensus protocol. Not only does this allow fast failover to maximize service availability, the algorithm ensures that only the most suitable secondary members are evaluated for election to primary and reduces the risk of unnecessary failovers (also known as "false positives"). Before a secondary replica is promoted, the election algorithms evaluate a range of parameters including:

- Analysis of election identifiers, timestamps, and journal persistence to identify those replica set members that have applied the most recent updates from the primary member.
- Heartbeat and connectivity status with the majority of other replica set members.
- User-defined priorities assigned to replica set members. For example, administrators can configure all replicas located in a remote region to be candidates for election only if the entire primary region fails.

Once the election process has determined the new primary, the secondary members automatically start replicating from it. When the original primary comes back online, it will recognize its change in state and automatically assume the role of a secondary, applying all write operations that occurred while it was down.

The number of replicas in a MongoDB replica set is configurable, with a larger number of replica members providing increased data durability and protection against database downtime (e.g., in case of multiple machine and regional failures, network partitions), or to isolate operational and analytical workloads running on the same cluster. Up to 50 members can be configured per replica set, providing operational flexibility and wide data distribution across multiple geographic sites, co-locating data in close proximity to remote users.

Extending flexibility, developers can configure replica sets to provide tunable, multi-node durability, and geographic awareness. For example, they can:

- Ensure write operations propagate to specific members of a replica set, deployed locally and in remote regions. MongoDB's write concern can be configured in such a

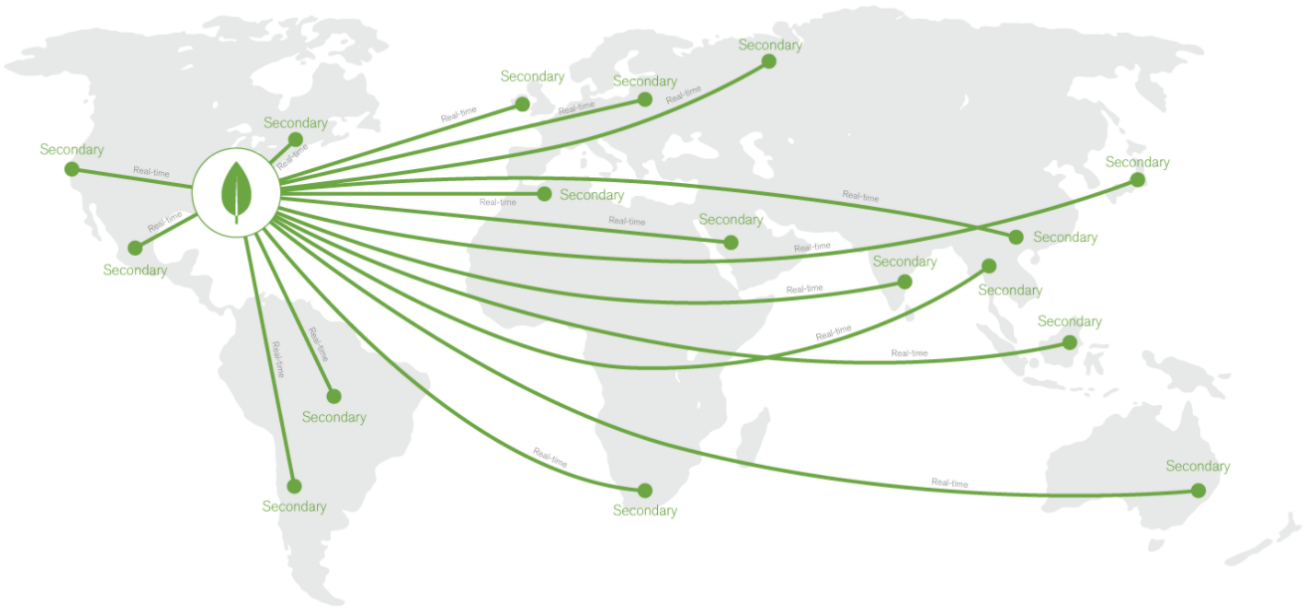

Figure 4: Replica sets enable global data distribution

way that writes are only acknowledged once specific policies have been fulfilled, such as writing to at least two replica set members in one region and at least one replica in a second region. This reduces the risk of data loss in the event of a complete data center outage.

- Ensure that specific members of a replica set respond to queries – for example, based on their physical location. The nearest read preference allows the client to read from the lowest-latency members of a replica set. This is typically used to route queries to a local data center, thus reducing the effects of geographic latency, while being able to immediately fallback to the next nearest replica if the closest node goes down. Tags can also be used to ensure that reads are always routed to a specific node or subset of nodes.

Workload Isolation

Beyond using replication for resilience and read scalability, replica sets also provide a foundation for combining different classes of workload on the same MongoDB cluster, each operating against its own copy of the data. With workload isolation, business analysts can run exploratory queries and generate reports, and data scientists can build machine learning models without impacting operational applications.

Within a replica set, one set of nodes can be provisioned to serve operational applications, replicating data in real time to other nodes dedicated to serving analytic workloads. By using MongoDB's native replication to move data in real time between the different node types, developers avoid lengthy and fragile ETL cycles, while analysts can improve both the speed and quality of insights and decision making by working with fresh, rather than aged and potentially stale data.

With the operational and analytic workloads isolated from one another on different replica set nodes, they never contend for resources. Replica set tags allow read operations to be directed to specific nodes within the cluster, providing physical isolation between analytics and operational queries. Different indexes can even be created for the analytics nodes, allowing developers to optimize for multiple query patterns. Data is exposed through MongoDB's rich query language, along with MongoDB Charts, and the Connector for BI for data visualization, and Connector for Spark to support real-time analytics.

Scalability

To meet the needs of apps with large data sets and high throughput requirements, MongoDB provides horizontal

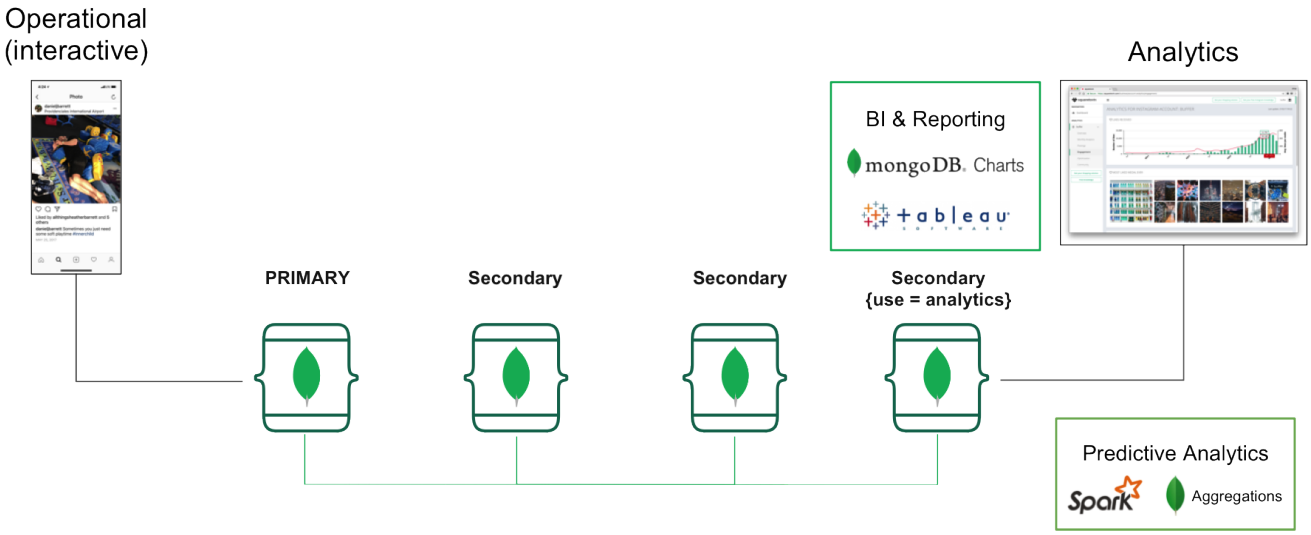

Figure 5: Combining operational and analytics workloads on a single data platform

scale-out for databases on low-cost, commodity hardware or cloud infrastructure using a technique called sharding. Sharding automatically partitions and distributes data across multiple physical instances called shards. Each shard is backed by a replica set to provide always-on availability and workload isolation. Sharding allows developers to seamlessly scale the database as their apps grow beyond the hardware limits of a single server, and it does this without adding complexity to the application. To respond to workload demand, nodes can be added or removed from the cluster in real time, and MongoDB will automatically rebalance the data accordingly, without manual intervention.

Sharding is transparent to applications; whether there is one or a thousand shards, the application code for querying MongoDB remains the same. Applications issue queries to a query router that dispatches the query to the appropriate shards. For key-value queries that are based on the shard key, the query router will dispatch the query to the shard that manages the document with the requested key. When using range-based sharding, queries that specify ranges on the shard key are only dispatched to shards that contain documents with values within the range. For queries that don't use the shard key, the query router will broadcast the query to all shards, aggregating and sorting the results as appropriate. Multiple query routers can be used within a MongoDB cluster, with the appropriate number governed by the performance and availability requirements of the application.

Unlike relational databases, MongoDB sharding is automatic and built into the database. Developers don't face the complexity of building sharding logic into their application code, which then needs to be updated as data is migrated across shards. They don't need to integrate additional clustering software or expensive shared-disk infrastructure to manage process and data distribution, or failure recovery.

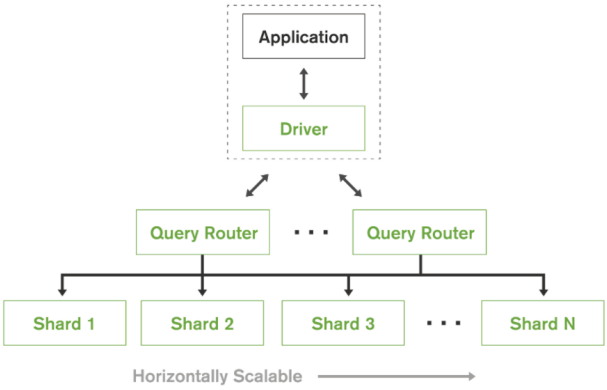

Figure 6: Automatic sharding for horizontal scale-out

By simply hashing a primary key value, many distributed databases randomly spray data across a cluster of nodes, imposing performance penalties when data is queried, or adding application complexity when data needs to be localized to specific nodes. By exposing multiple sharding policies to developers, MongoDB offers a better approach. Data can be distributed according to query patterns or data

placement requirements, giving developers much higher scalability across a diverse set of workloads:

- **Ranged Sharding**. Documents are partitioned across shards according to the shard key value. Documents with shard key values close to one another are likely to be co-located on the same shard. This approach is well suited for applications that need to optimize range based queries, such as co-locating data for all customers in a specific region on a specific shard.
- **Hashed Sharding**. Documents are distributed according to an MD5 hash of the shard key value. This approach guarantees a uniform distribution of writes across shards, which is often optimal for ingesting streams of time-series and event data.
- **Zoned Sharding**. Provides the ability for developers to define specific rules governing data placement in a sharded cluster. Zones are discussed in more detail in the following Data Locality section of the guide.

Thousands of organizations use MongoDB to build high-performance systems at scale. You can read more about them on the MongoDB scaling page.

Data Locality

MongoDB zoned sharding allows precise control over where data is physically stored in a cluster. This allows developers to accommodate a range of application needs – for example controlling data placement by geographic region for latency and governance requirements, or by hardware configuration and application feature to meet a specific class of service. Data placement rules can be continuously refined by modifying shard key ranges, and MongoDB will automatically migrate the data to its new zone.

The most popular use cases for MongoDB zones include the following:

Geographic Data Placement

MongoDB gives developers the ability to create zones in multiple geographic regions. Each zone is part of the same, single cluster and can be queried globally, but data is pinned to shards in specific regions based on data locality requirements. Developers simply name a shard by region, tag their documents by region in the shard key, and MongoDB does the rest.

By associating data to shards based on regional policies, developers can create global, always-on, write-everywhere clusters, with each shard serving operations local to it – enabling the database to serve distributed, write-heavy workloads with low latency. This design brings the benefits of "multi-master" database, without introducing the complexity of eventual consistency or data loss caused by conflicting writes.

Zoned sharding also enables developers to keep user data within specific regions to meet governance requirements for data sovereignty, such as the EU's GDPR. To illustrate further, an application may have users in North America, Europe, and China. The developer can assign each shard to a zone representing the physical location (North America, Europe, or China) of that shard's servers, and then map all documents to the correct zone based on its region field. Any number of shards can be associated with each zone, and each zone can be scaled independently of the others – for instance, accommodating faster user growth in China than North America. Zones are available as a part of Global Clusters in the MongoDB Atlas managed database service, discussed later in this guide.

Learn more by reviewing our tutorial on creating geographically distributed clusters with MongoDB zoned sharding.

Class of Service

Data for a specific application feature or customer can be associated with specific zones. For instance, a company offering Software-as-a-Service (SaaS) may assign users on its free usage tier to shards provisioned on lower specified hardware, while paying customers are allocated to premium infrastructure. The SaaS provider has the flexibility to scale parts of the cluster differently for free users and paying customers. For example, the free tier can be allocated just a few shards, while paying customers can be assigned to dozens of shards.

Learn more by reviewing our tutorial on configuring application affinity with MongoDB zoned sharding.

Building upon application features, zoned sharding also enables deployment patterns such as tiered, or

multi-temperature storage. Different subsets of data often have different response time requirements, usually based on access frequency and age of the data. For example, IoT applications or social media services handling time-series data will demand that users experience the lowest latency when accessing the latest data. This data can be pinned to the highest performance hardware with fast CPUs and SSDs. Meanwhile, aged data sets that are read less frequently typically have relaxed latency SLAs, so can be moved onto slower, less expensive hardware based on conventional, high capacity spinning disks. By including a timestamp in the shard key, the MongoDB cluster balancer can migrate data based on age from the high-performance tier to the active archive tier.

Learn more by reviewing our tutorial on configuring tiered storage with MongoDB zoned sharding

Data Security

Having the freedom to put data where it's needed enables developers to build powerful new classes of application. However, they must also be confident that their data is secure, wherever it is stored. Rather than build security controls back in the application, they should be able to rely on the database to implement the mechanisms needed to protect sensitive data and meet the needs of apps in regulated industries.

MongoDB features extensive capabilities to defend, detect, and control access to data:

- **Authentication.** Simplifying access control to the database, MongoDB offers integration with external security mechanisms including LDAP, Windows Active Directory, Kerberos, and x.509 certificates. In addition, IP whitelisting allows DevOps teams to configure MongoDB to only accept external connections from approved IP addresses.
- **Authorization**. Role-Based Access Controls (RBAC) enable DevOps teams to configure granular permissions for a user or an application based on the privileges they need to do their job. These can be defined in MongoDB, or centrally within an LDAP server. Additionally, developers can define views that expose only a subset of data from an underlying collection, i.e. a view that filters or masks specific fields, such as Personally Identifiable Information (PII) from customer data or health records. Views can also be created to only expose aggregated data.
- **Auditing.** For regulatory compliance, security administrators can use MongoDB's native audit log to track any database operations – whether DML or DDL.
- **Encryption.** MongoDB data can be encrypted on the network, on disk and in backups. With the Encrypted storage engine , protection of data-at-rest is an integral feature within the database. By natively encrypting database files on disk, developers eliminate both the management and performance overhead of external encryption mechanisms. Only those staff who have the appropriate database authorization credentials can access the encrypted data, providing additional levels of defense.

To learn more, download the MongoDB Security Reference Architecture Whitepaper.

MongoDB Mobile

Available in beta, MongoDB Mobile extends your ability to put data where you need it, all the way out to the edge of the network on IoT assets and iOS and Android mobile devices. MongoDB Mobile provides a single database, query language, and the intuitive Stitch SDK that runs consistently for data held on mobile clients, through to the backend server.

MongoDB Mobile provides the power and flexibility of MongoDB in a compact form that is power and performance aware with a low disk and memory footprint. It supports 64 bit iOS and Android operating systems and is easily embedded into mobile and IoT devices for fast and reliable local storage of JSON documents. With secondary indexing, access to the full MongoDB query language and aggregations, users can query data any way they want. With local reads and writes, MongoDB Mobile lets you build the fastest, most reactive apps. Stitch Mobile Sync (coming soon) will automatically synchronize data changes between data held locally and your backend database, helping resolve any conflicts – even after the device has been offline.

The beta program is open now, and you can sign up for access on the MongoDB Mobile product page.

Freedom to Run Anywhere

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.

However, relational databases that have been designed to run on a single server are architecturally misaligned with modern cloud platforms, which are built from low-cost commodity hardware and designed to scale out as more capacity is needed. For example, cloud applications with uneven usage or spikes during certain periods require built-in elasticity and scalability across the supporting technology stack. Legacy relational databases do not natively support these capabilities requiring teams to try and introduce distributed systems properties through approaches such as application-level sharding.

It's for this reason that modern, non-tabular databases delivered as a service are growing in popularity amongst organizations moving into the cloud. But many of these database services run exclusively in a single cloud platform, which increases business risk. For the past decade, more companies are adopting open source technologies to reduce lock-in with proprietary vendors. Choosing to build applications on a proprietary cloud database re-introduces the risk of lock-in, this time to cloud vendor APIs and technologies that only run in a single environment.

To reduce the likelihood of cloud lock-in, teams should build their applications on distributed databases that will deliver a consistent experience across any environment. As an open source database, MongoDB can be deployed anywhere — from mainframes to a private cloud to the public cloud. The developer experience is entirely unaffected by the deployment model; similarly, teams responsible for standing up databases, maintaining them, and optimizing performance can also leverage a unified set of tools that deliver the same experience across different environments.

MongoDB allows organizations to adopt cloud at their own pace by moving select workloads as needed. For example, they may run the same workload in a hybrid environment to manage sudden peaks in demand, or use the cloud to launch services in regions where they lack a physical data center presence.

MongoDB Atlas

Similar to the way MongoDB and Stitch dramatically improve developer productivity, MongoDB offers a fully managed, on-demand and elastic service, called MongoDB Atlas, 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. Atlas allows customers to adopt a more agile, on-demand approach to IT rather than underutilizing cloud as merely a hosted infrastructure platform and replicating many of the same operational, administrative, and time-to-market challenges with running on-premises. Built-in automation and proven best practices reduce the likelihood of human error and minimize operational overhead. Key features of MongoDB Atlas include:

Automation and elasticity. MongoDB Atlas automates infrastructure provisioning, setup, and deployment so teams can get the database resources they need, when they need them. Patches and minor version upgrades are applied automatically. Database modifications — whether it's to scale out or perform an upgrade — can be executed in a few clicks or an API call with no downtime window required.

High availability and durability. MongoDB Atlas automatically creates self-healing, geographically distributed clusters with a minimum of 3 nodes to ensure no single point of failure. MongoDB Atlas also includes powerful features to enhance reliability for mission-critical production databases, such as fully managed backups with point-in-time recovery and queryable snapshots, which allow customers to restore granular data sets in a fraction of the time it would take to restore an entire snapshot.

Global clusters Global cluster support enables the simplified deployment and management of a single

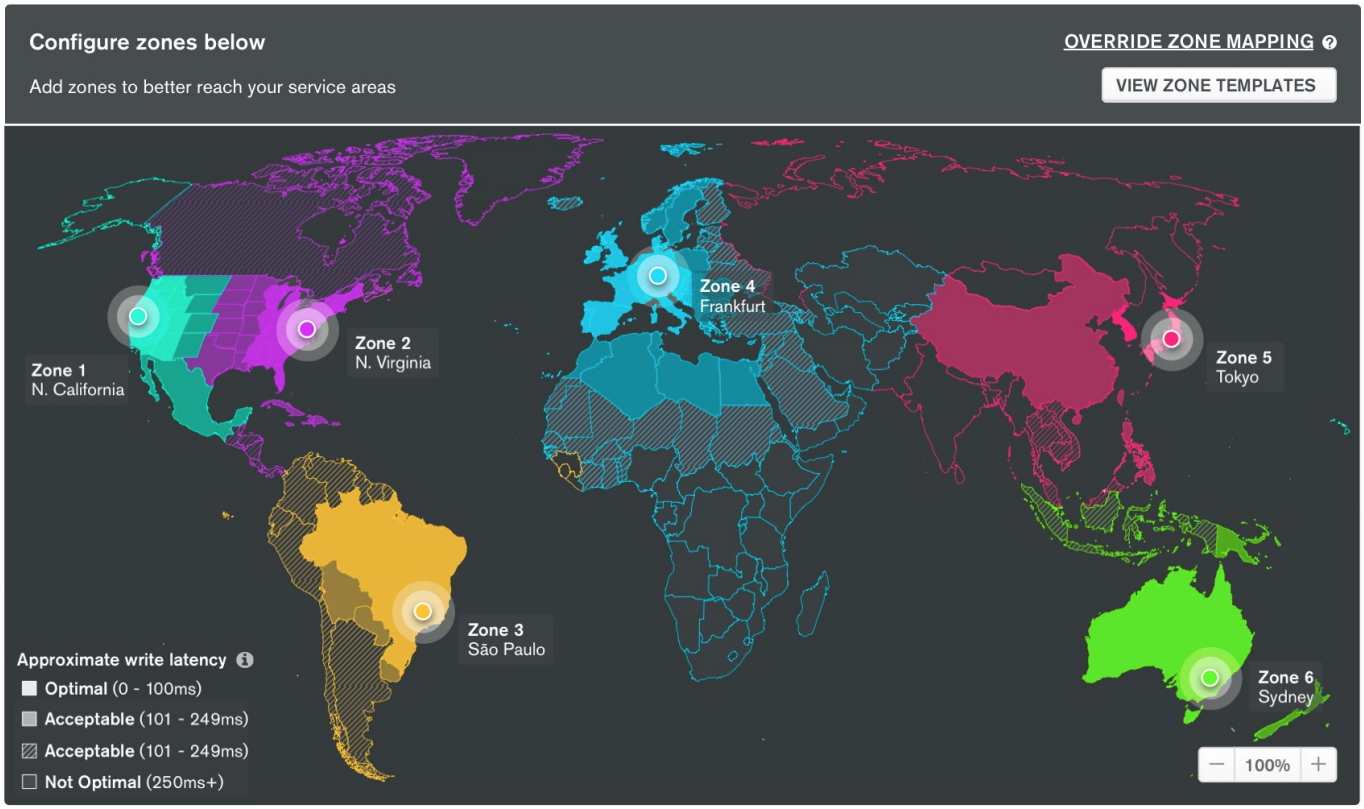

Figure 7: Serving always-on, globally distributed, write-everywhere apps with MongoDB Atlas Global Clusters and zoned sharding

geographically distributed database. Organizations can easily control where data is distributed to allow low-latency reads and writes and meet the data sovereignty requirements demanded by new data privacy regulations. Data can also be easily replicated and geographically distributed, enabling multi-region fault tolerance and fast, responsive reads to any dataset, from anywhere.

Secure by default. MongoDB Atlas makes it easy for organizations to control access to their managed databases by automatically incorporating many of the security features mentioned earlier in this architecture guide. For example, a customer's database instances are deployed with robust access controls and end-to-end encryption. Other security features include network isolation, IP whitelisting, VPC peering, and always-on authentication.

Additional enterprise security features allow organizations to exercise more control by enabling integration with existing security platforms. Customers can use their own LDAP servers to simplify access control and granular permissions management for users and applications. They can manage their own encryption keys for data at rest by integrating with their key management services. And finally, security administrators can enable database-level auditing to track any operation taken against the database.

Comprehensive monitoring and performance optimization. MongoDB Atlas includes an integrated set of features that simplify database monitoring and performance optimization. Developers can get deep visibility into their clusters using optimized charts tracking dozens of key metrics, and easily customize and send alerts to channels such as Slack, Datadog, and PagerDuty. MongoDB Atlas also allows customers to see what's happening in their clusters as it happens with the Real-Time Performance Panel, and allows them to take advantage of automatically generated index suggestions via the built-in Performance Advisor to improve query performance. Finally, the built-in data explorer is a convenient way for users to run queries, view metadata about their collections, monitor index usage, and interact with their data with full CRUD functionality.

Live migration. MongoDB Atlas makes it easy to migrate live data from MongoDB deployments running in any other environment. Atlas will perform an initial sync between the migration destination and the source database, and use the oplog to keep the two database in sync until teams are prepared to perform the cutover process. Live migration supports importing data from replica sets, sharded clusters, and any deployment running MongoDB 2.6 or higher.

Widespread coverage on the major cloud platforms. MongoDB Atlas is available in over 50 cloud regions across Amazon Web Services, Microsoft Azure, and Google Cloud Platform. Organizations with a global user base can use MongoDB Atlas to automatically replicate data to any number of regions of their choice to deliver fast, responsive access to data wherever their users are located. Furthermore, unlike other open source database services which vary in terms of feature-support and optimizations from cloud provider to cloud provider, MongoDB Atlas delivers a consistent experience across each of the cloud platforms, ensuring developers can deploy wherever they need to, without compromising critical functionality.

You can learn about MongoDB Atlas and all of the features discussed above in the documentation. And you can take Atlas for a spin at no cost on the free tier.

MongoDB Ops Manager

For organizations that need to run the database on their own self-managed infrastructure for business or regulatory requirements, MongoDB offers SaaS or on-premises management tools available that enable customers to build their own MongoDB service for internal development teams.

MongoDB Ops Manager is the simplest way to run MongoDB on premises or in a private cloud, making it easy for operations teams to deploy, monitor, backup, and scale MongoDB. The capabilities of Ops Manager are also available in the MongoDB Cloud Manager tool, delivered as SaaS in the cloud.

Deployments and upgrades. Whereas MongoDB Atlas is a fully managed database as a service platform, Ops Manager provides a powerful suite of tools that enable operations teams to implement and automate MongoDB deployment and maintenance tasks in accordance with their policies and best practices. Ops Manager coordinates critical operational tasks across the servers in a MongoDB system. It communicates with the infrastructure through agents installed on each server. The servers can reside in the public cloud or a private data center. Ops Manager reliably orchestrates the tasks that administrators have traditionally performed manually – deploying a new cluster, performing upgrades, creating point-in-time backups, and many other operational activities.

Ops Manager also makes it possible to dynamically resize capacity by adding shards and replica set members. Other maintenance tasks such as upgrading MongoDB, building new indexes across replica sets or resizing the oplog can be reduced from dozens or hundreds of manual steps to the click of a button, all with zero downtime. Administrators can use the Ops Manager interface directly, or invoke the Ops Manager RESTful API from existing enterprise tools.

Cloud Native Integration. Ops Manager can be integrated with Pivotal Cloud Foundry, Red Hat OpenShift, and Kubernetes. With Ops Manager, you can rapidly deploy MongoDB Enterprise powered applications by abstracting away the complexities of managing, scaling and securing hybrid clouds. Ops Manager coordinates orchestration between your cloud native platform, which handles the underlying infrastructure, while Ops Manager handles the MongoDB instances, automatically configured and managed with operational best practices.

With this integration, you can consistently and effortlessly run workloads wherever they need to be, standing up the same database configuration in different environments, all controlled from a single pane of glass.

Ops Manager features such as server pooling make it easier to build a database as a service within a private cloud environment. Ops Manager will maintain a pool of globally provisioned servers that have agents already installed. When users want to create a new MongoDB deployment, they can request servers from this pool to host the MongoDB cluster. Administrators can even associate certain properties with the servers in the pool and expose server properties as selectable options when a user initiates a request for new instances.

Comprehensive monitoring and performance optimization The monitoring, alerting, and performance

optimization capabilities of Ops Manager and Cloud Manager are similar to what's available with MongoDB Atlas. Integration with existing monitoring tools is straightforward via the Ops Manager and Cloud Manager RESTful API, and with packaged integrations to leading Application Performance Management (APM) platforms, such as New Relic. These integrations allow MongoDB status to be consolidated and monitored alongside the rest of your application infrastructure, all from a single pane of glass.

Disaster Recovery: Backups & point-in-time recovery
Similar to how backups are handled in MongoDB Atlas, Ops Manager and Cloud Manager backups are maintained continuously, just a few seconds behind the operational system. Because Ops Manager reads the oplog used for replication, the ongoing performance impact is minimal – similar to that of adding an additional replica to a replica set. If the MongoDB cluster experiences a failure, the most recent backup is only moments behind, minimizing exposure to data loss. Ops Manager and Cloud Manager both also offer point-in-time backup of replica sets and cluster-wide snapshots of sharded clusters. Users can restore to precisely the moment they need, quickly and safely. Automation-driven restores allows a fully configured cluster to be re-deployed directly from the database snapshots in a just few clicks. Similar to MongoDB Atlas, Ops Manager and Cloud Manager also provide the ability to query backup snapshots.

Ops Manager can also be deployed to control backups to a local data center or AWS S3. If using Cloud Manager, customers receive a fully managed backup solution with a pay-as-you-go model. Dedicated MongoDB engineers monitor user backups on a 24x365 basis, alerting operations teams if problems arise.

Cloud Adoption Stages

By building on a database that runs the same across any environment and using an integrated set of management tooling that delivers a consistent experience across the board, organizations can ensure a seamless journey from on-premises to the public cloud:

- Teams dipping their toe into the cloud can start with MongoDB on premises and optimize ongoing management using Ops Manager. Through integration with OpenShift and Cloud Foundry, Ops Manager can be used as a foundation for your own private cloud database service.
- As their level of comfort with the public cloud increases, they can migrate a few deployments and self-manage using Cloud Manager or try the fully managed, on-demand, as-a-service approach with MongoDB Atlas.
- Cloud-first organizations interested in exploiting the benefits of a multi-cloud strategy can use MongoDB Atlas to easily spin up clusters and replicate data across regions and cloud providers, all without worrying about operations or platform lock-in.

Conclusion and Next Steps

Every industry is being transformed by data and digital technologies. As you build or remake your company for a digital world, **speed matters** – measured by how fast you

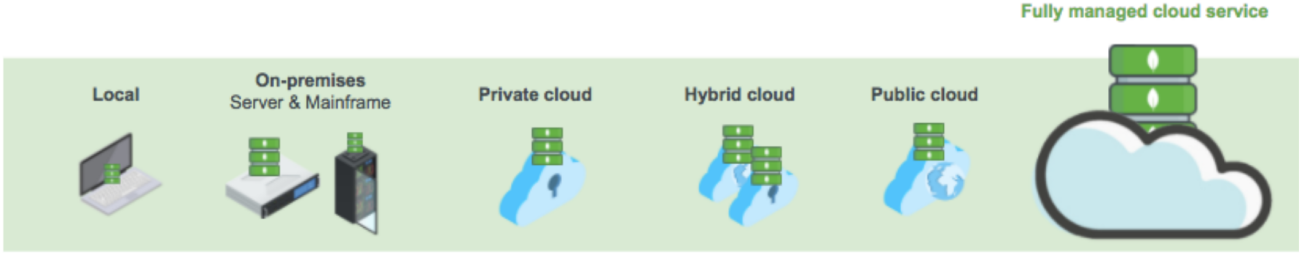

Figure 8: MongoDB provides you the freedom to run anywhere

build apps, how fast you scale them, and how fast you can gain insights from the data they generate. These are the keys to applications that provide better customer experiences, enable deeper, data-driven insights or make new products or business models possible.

With its intelligent operational data platform, MongoDB enables developers through:

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

In this guide we have explored the fundamental concepts that underpin the architecture of MongoDB. Other guides on topics such as performance, operations, and security best practices can be found at mongodb.com.

You can get started now with MongoDB by:

1. Spinning up a fully managed MongoDB instance on the Atlas free tier
2. Downloading MongoDB for your own environment
3. Reviewing the MongoDB manuals and tutorials on our documentation page

We Can Help

We are the MongoDB experts. Over 5,700 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)

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