# Mongo DB Architecture

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 new classes of 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.

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.

### The Best Way to Work with Data: The Document Model

Relational 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.

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 relational 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
* **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

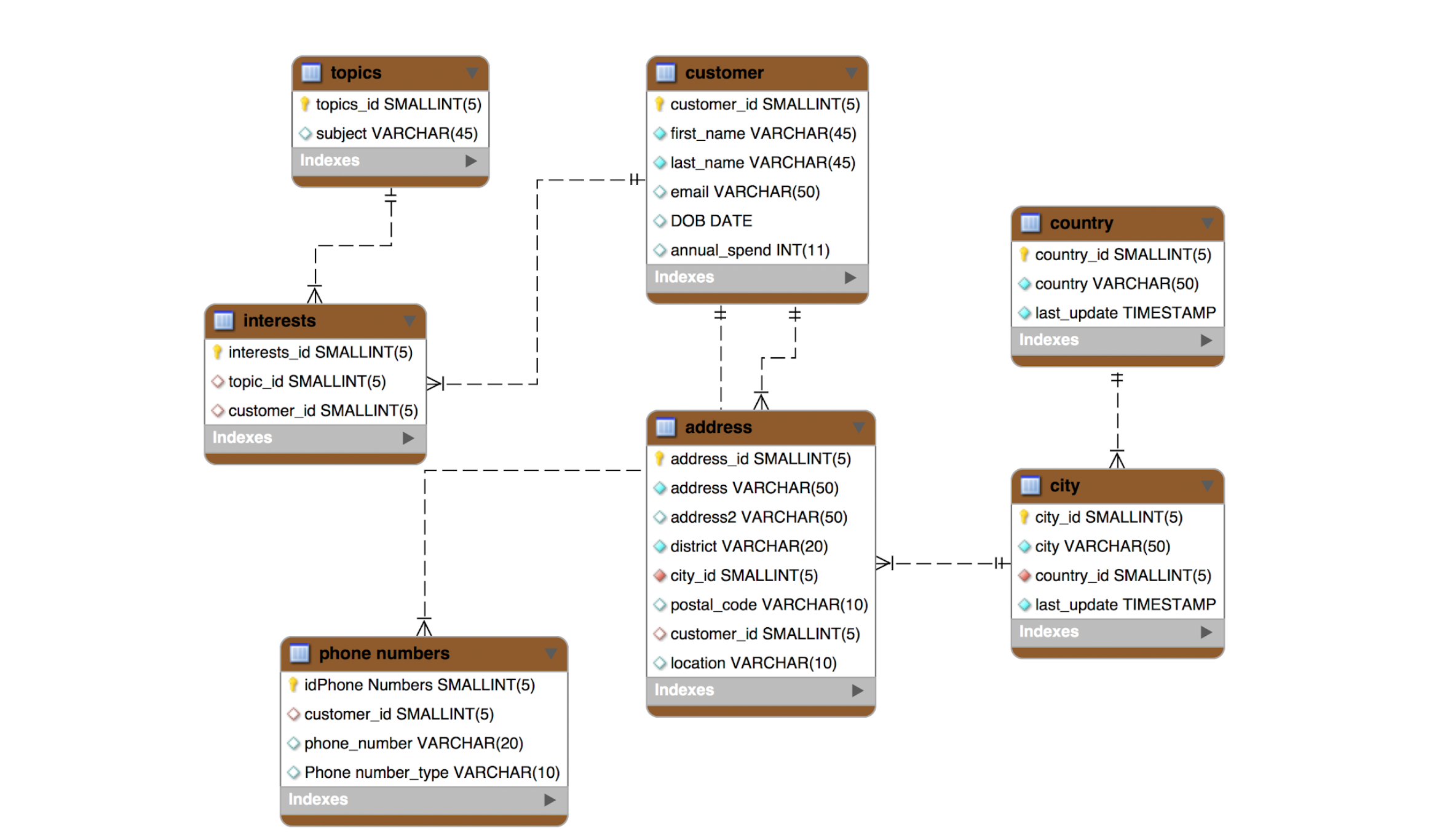
MongoDB’s document model delivers these benefits for developers, making it the best way to work with data.

### Easy: A Natural, Intuitive Data Model

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, 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.

{

"name":{

"first":"Eartha",

"second":"Thompson"

},

"address":[

{

"location":"home",

"number":23,

"street":"Twin Pines",

"city":"New York",

"state":"New York",

"postalCode":"O83 1F1"

},

{

"location":"work",

"number":1,

"street":"Holy Cross",

"city":"New York",

"state":"New York",

"postalCode":"513 8U5"

}

],

"phone":[

{

"location":"mobile",

"number":"+48-675-560-3029"

},

{

"location":"work",

"number":"+48-887-222-1234"

}

],

"email":"eandrzejak0@yellowpages.com",

"annualSpend":916305.32,

"dob":"1985-02-28 07:32:58",

"interests":[

{

"interest":"XML Schema Design"

},

{

"interest":"Glazing"

}

]

}

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 developers to 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, run ad hoc queries and evaluate their performance, view and create indexes, build data validation rules, and more. Compass provides an intuitive interface for working with MongoDB.

### Flexibile: 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 structure 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 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 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 into 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 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.

### Fast: 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. 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 between multiple collections. This is very useful for analytics workloads, but is generally not required for operational use cases.

This 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 and SQL to insert a single user record, where users have multiple properties including name, all of their addresses, phone numbers, interests, and more.

****

**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.

#### Versatile: Various Data Models and Access Patterns

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 – 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.

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.

##### Data Consistency Guarantees

MongoDB’s versatility also 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 below). 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.

##### Transactional Model

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

The addition of multi-document transactions, scheduled for MongoDB 4.0, makes it even easier for developers to address more use cases with MongoDB. They feel just like the transactions, developers are familiar with from relational databases – multi-statement, similar syntax, and easy to add to any application. Through snapshot isolation, transactions provide a globally consistent view of data, enforce all-or-nothing execution, and will not impact performance for workloads that do not require them.

##### MongoDB Stitch

MongoDB Stitch, Serverless for data-driven applications. Stitch streamlines application development with simple, secure access to data and services from the client – getting your apps to market faster while reducing operational costs. Stitch provides full access to your MongoDB database, in addition to public cloud services – all through an intuitive SDK. Add business logic to your backend using Stitch's hosted functions. Take advantage of Stitch's HTTP service and Webhooks to integrate with your microservices and provide secure APIs. Stitch secures access to data, services, and functions through powerful, declarative rules – putting you in control.

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. That 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.

#### 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.

* **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).

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:

* 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.
* 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, 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.

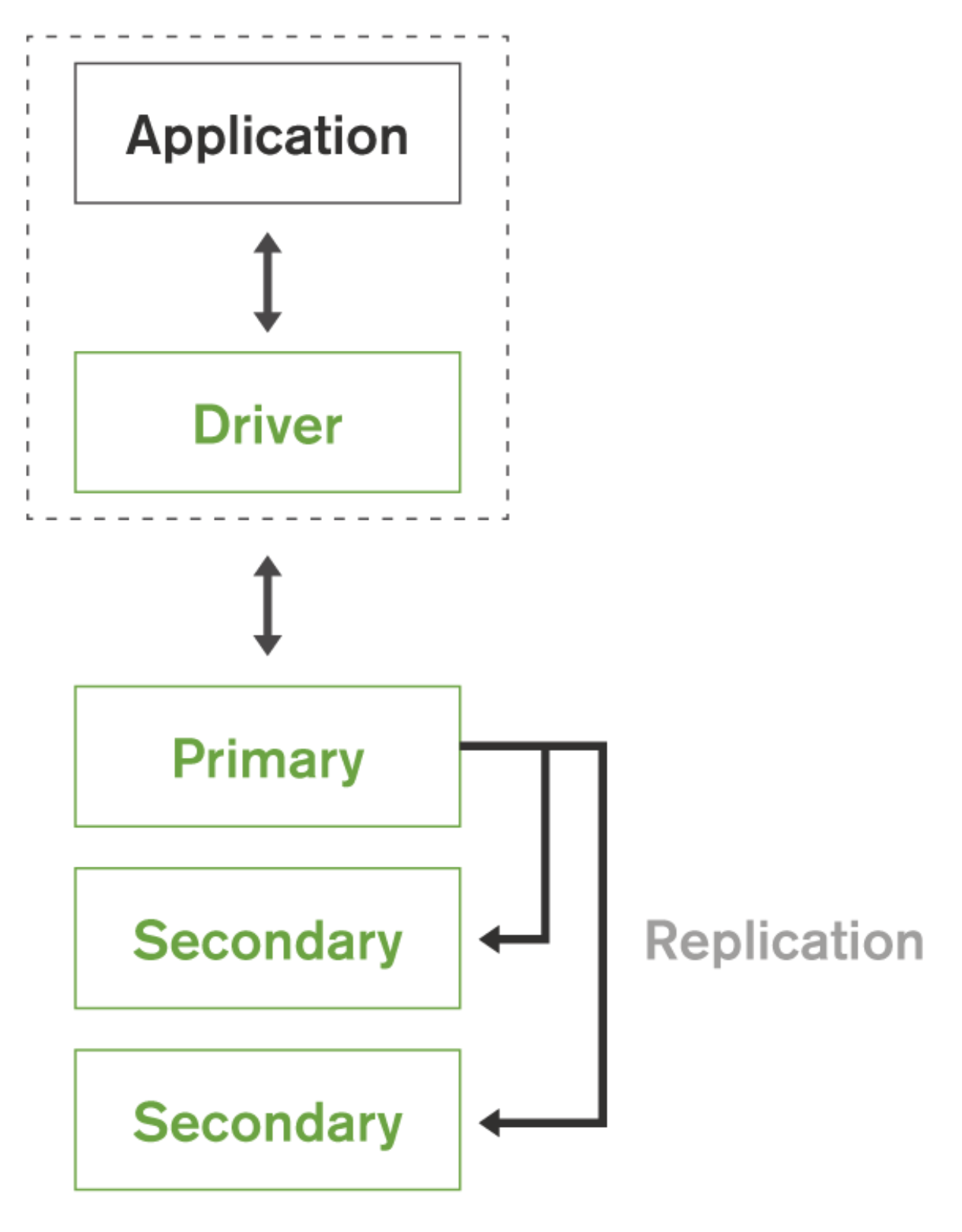
#### MongoDB Distributed Systems Architecture

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

* Availability
* Workload isolation
* Scalability
* Data locality

##### Availability

MongoDB maintains multiple copies of data using replica sets (Figure 3). Unlike relational databases, replica sets are self-healing as failover and recovery are 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 an outage (e.g., a power failure, 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 have occurred during its outage.

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 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 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.

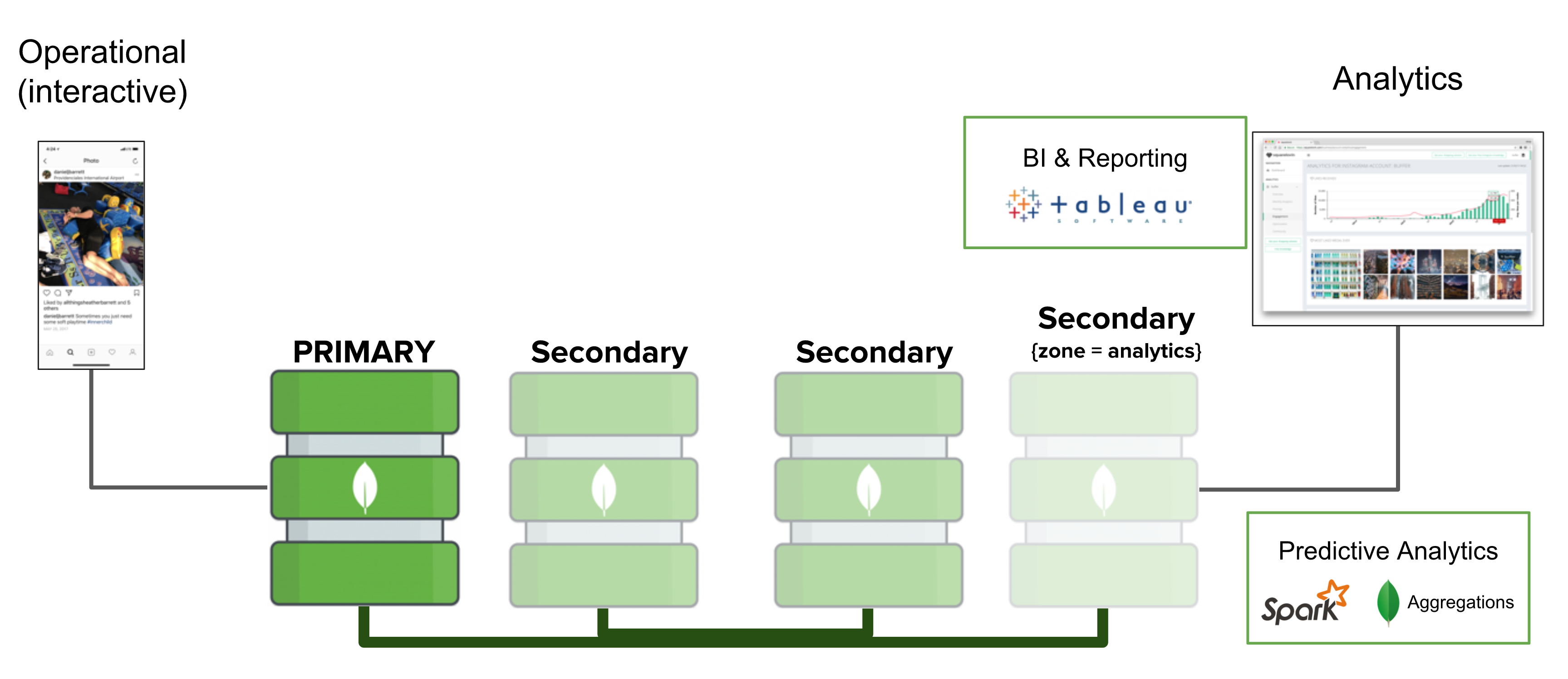
##### Workload Isolation

Beyond using replication for redundancy and availability, 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 the Connector for BI and Connector for Spark to support real-time analytics and data visualization.

**Figure 4:** Replica sets enable global data distribution

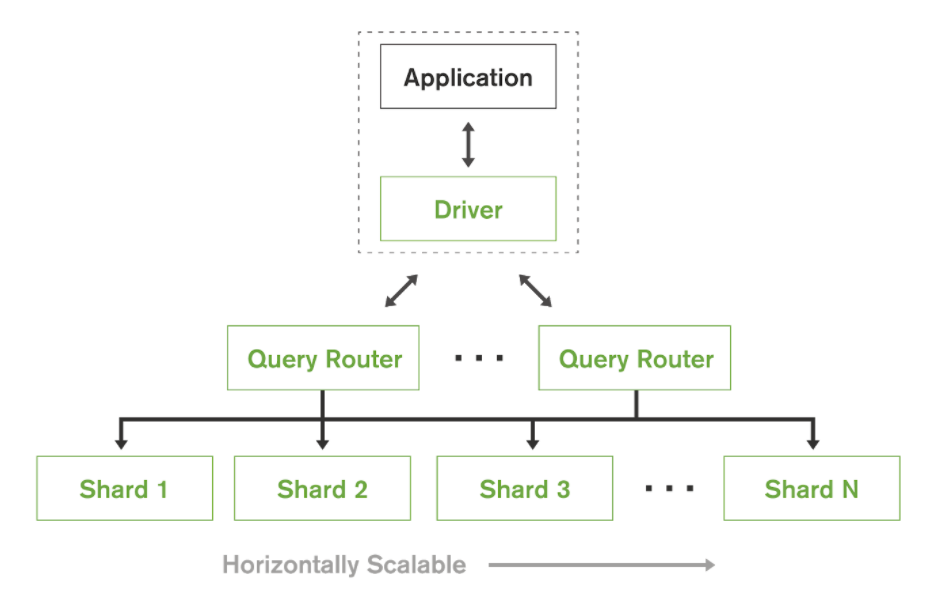
**Figure 5:** Combining operational and analytics workloads on a single data platform

##### Scalability

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

**Figure 6:** Automatic sharding for horizontal scale-out

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.

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 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.

##### 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.

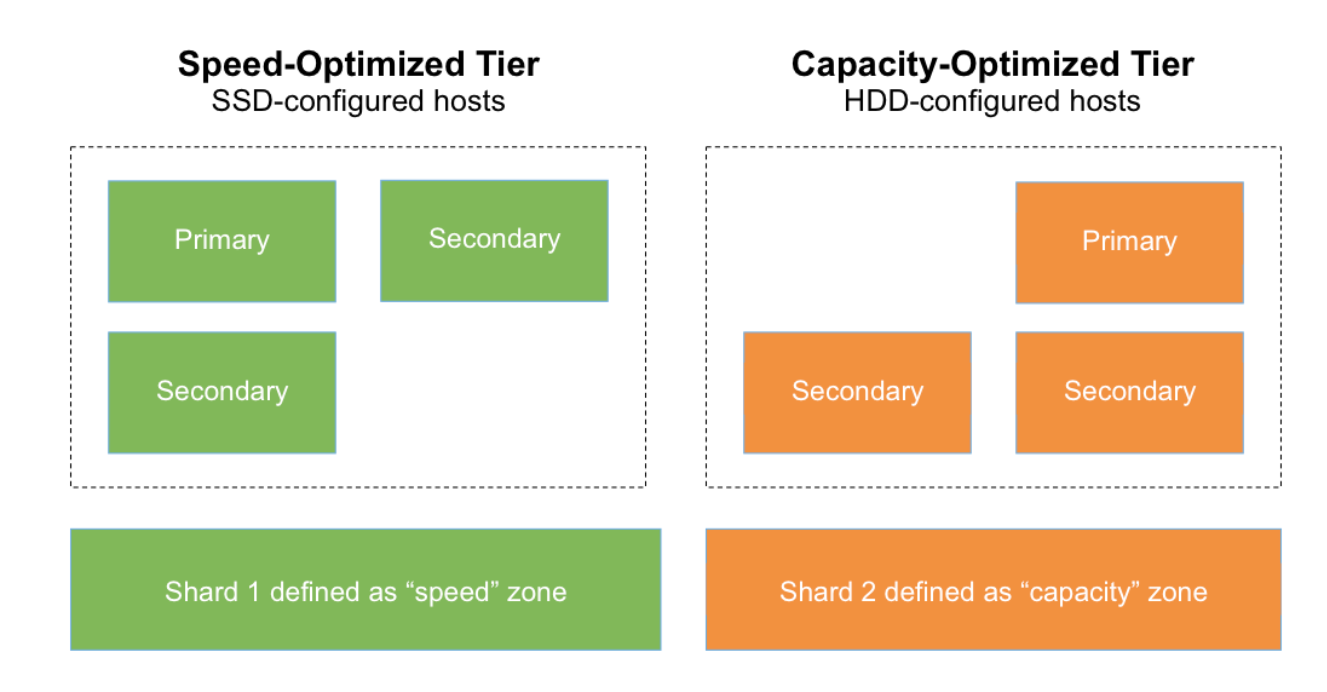
By associating data to shards based on regional policies, developers can create global, always-on, write-everywhere clusters, with each shard serving operations local to it – enabling the database to serve distributed, write-heavy workloads with low latency. This design brings the benefits of “multi-master” 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.

##### 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.

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.

**Figure 7:** Implementing 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.

#### 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 on-demand services that make it easier to build and run an application backend. This move from building IT to consuming IT as a service is well aligned with a parallel organizational shift happening across companies prioritizing productivity and getting to market faster — a move from specialized and often siloed groups to more cross-functional, DevOps teams that are able to make many of their own technology decisions. The result is often a far more nimble and focused organization that is able to rapidly deliver new digital products using agile methodologies and modern application architectures, such as microservices.

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, companies have increasingly adopted 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 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. Even better availability guarantees are possible by enabling cross-region replication to achieve multi-region fault tolerance. MongoDB Atlas also includes powerful features to enhance reliability for mission-critical production databases, such as continuous, incremental 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.
* **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, always-on authentication, and much more.
* **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 lets operations teams run queries to review document structure and database schema, view collection metadata, and inspect index usage statistics.
* **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.

#### 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 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:

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

**Figure 8:** MongoDB provides you the freedom to run anywhere