MongoDB Evolved

Becoming the World’s “Most Wanted” Database. March 2021
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>Data Integrity and Resilience</td>
<td>2</td>
</tr>
<tr>
<td>Performance and Scalability</td>
<td>6</td>
</tr>
<tr>
<td>Expansion of the MongoDB Query API &amp; Drivers</td>
<td>8</td>
</tr>
<tr>
<td>Security and Privacy</td>
<td>12</td>
</tr>
<tr>
<td>Conclusion</td>
<td>14</td>
</tr>
<tr>
<td>We Can Help</td>
<td>14</td>
</tr>
<tr>
<td>Resources</td>
<td>15</td>
</tr>
</tbody>
</table>
Introduction

The first production release of MongoDB shipped in August 2009. The 1.0 version and releases that followed over the next several years were focused on validating a new and largely unproven approach to database design – built on a document data model and layered onto an elastic and distributed systems foundation.

Why was this new approach necessary? Because the founders of MongoDB, along with developers from the community they worked with, had been struggling to meet the demands of building modern applications on top of decades-old relational databases. They were constrained by:

1. **Rigid, tabular schemas** that slowed them down. This was because the schema bore no resemblance to the objects they worked with in code, and it couldn’t be easily adapted as their applications evolved.

2. A **monolithic, scale-up architecture** that was at odds with the scale-out, cloud-native infrastructure that new applications were being run on.

Those early MongoDB releases attracted masses of industry attention and real-world adoption across startups and enterprises alike. Organizations as diverse as Bosch, Buzzfeed, CERN, Cisco, Codecademy, Disney, Electronic Arts, Forbes, Foursquare, Github, O2, Shutterfly, Sourceforge, Stripe, and many more started using MongoDB.

With early usage validating product/developer fit, the MongoDB engineering team’s focus shifted to doing two things:

1. **Industrializing the database** for use by the wider developer community.

2. Serving a **broader range of use cases**.

That focus has driven major investments in a number of critical areas, including database resilience and scalability, the versatility of the MongoDB query API and drivers, and in expanding security and privacy controls.

Beyond the core database, the engineering team also invested in delivering MongoDB not just as software you could download and run yourself, but also as a fully-managed service in the cloud, freeing developers from the overhead of operating the database themselves. The
The following section of the paper charts the evolution of each of these areas.

Data Integrity and Resilience

MongoDB’s core mechanisms for data integrity are founded on:

- Read and write concerns.
- A global logical clock, used to synchronize operations across the database cluster.
- ACID transactional guarantees.
- Schema management.

Building on data integrity, resilience is enabled primarily through:

- Replica sets.
- The underlying replica set election protocol.
- Retryable reads and writes.

Read and Write Concerns

Unlike traditional relational databases, MongoDB is a distributed system. This difference introduces concepts that will be new for developers who are using MongoDB for the first time. Within MongoDB, data is automatically replicated across multiple independent nodes in a replica set for redundancy and resilience, and can be partitioned across shards for horizontal scale-out.

Within a cluster, MongoDB uses a combination of write concerns to control data durability and read concerns to control consistency and isolation. Read and write concerns are fully tunable, giving developers the flexibility to balance latency and throughput against the application’s desired SLAs. You can configure read and write concerns both client-side in the MongoDB drivers, and from MongoDB 4.4 released in 2020, centrally in the database using a global, cluster-wide default.

Early versions of MongoDB shipped with a default "unacknowledged" (w:0) write concern. This default was used because originally the MongoDB database was designed as a backend to an application PaaS (Platform-as-a-Service) that would enforce durability, and wasn’t going to be exposed directly to developers. The default was changed back in November 2012 to the stronger “acknowledged” (w:1) write concern that waits for the write operation to be applied to database memory before acknowledging success back to the application. This change of default followed the addition of an on-disk journal (write-ahead log) to MongoDB 1.8 in 2011.

Beyond MongoDB’s default write concern, developers can further dial-up durability by configuring the write operation to only return success to the application once specific policies have been satisfied:

- The write has been persisted to the MongoDB node’s on-disk journal, using the j: true option. This provides data safety if the node crashes before checkpoints have been applied from memory to disk, and allows faster recovery when the node comes back up.
- The write has been acknowledged by a majority of replicas (w: majority) in the cluster.
The write has been acknowledged by a user-defined number of replicas. Custom write concerns allow you to define specific durability behavior, such as only acknowledging success to the application once a write has been applied to a replica in each region across a globally distributed cluster. This custom write concern provides write safety in the event of a complete regional outage.

Write concern behavior has been further enhanced over the years:

- With MongoDB 3.2 released in 2015 the majority write concern was hardened to ensure each operation was committed to both the memory and journal on a majority of replicas before acknowledging success back to the application. This ensures that data cannot be rolled back in the event that a new primary is elected before the write has replicated to a majority of the replicas. It also makes the data “crash safe” should a node fail completely.
- From May 2019, the MongoDB Atlas default connection string used by the drivers to the database was upgraded to the majority write concern. This has provided the MongoDB engineering team with valuable insight into changing the database default to use majority writes in the next major release of the database.
- The introduction of Streaming Replication in 2020 (discussed later), and by replicating data before it has been journaled, has reduced the latency of majority committed writes by up to 50%. This makes performance overhead of the stricter guarantee imperceptible to most users.

Moving on to read operations, MongoDB has long offered read preferences, enabling developers to configure the drivers to route read requests to specific replica set nodes. This enables developers to control read availability and latency, and to isolate operational from analytical queries across separate replicas.

To configure read consistency and isolation MongoDB 3.2 introduced the readConcern query option. The driver can pass a majority readConcern to only return data that has been written to a majority of nodes, and which therefore cannot be rolled back after a replica set election. By default, MongoDB uses a readConcern of "local" to return the most recent data available on the replica at the time of the query.

Like write concerns, MongoDB’s read isolation and consistency levels have been enhanced over time:

- The linearizable read concern was introduced with MongoDB 3.4 in 2016. This enables multiple threads to perform reads and writes on a single document as if a single thread performed each operation sequentially. The query waits for concurrently executing writes to propagate to a majority of replica set members before returning results. The linearizable read concern provides the strongest level of read isolation at the expense of higher latency.
- With MongoDB 3.6 in 2017, MongoDB added support for causal consistency – guaranteeing that every read operation within a client session will always see the previous write operation, regardless of which replica is serving the request. This capability was enabled by the introduction of a global logical clock to MongoDB, which is discussed in the section below. By enforcing strict, causal ordering of operations within a session, causal consistency ensures every read is always logically consistent, enabling monotonic reads from a distributed system – a guarantee that cannot be met by most distributed databases.
- The snapshot read concern was added with the introduction of multi-document ACID transactions in 2018 (discussed later). It enables point-in-time reads to operate against a globally consistent snapshot of the database across each primary node of a sharded cluster. By querying the snapshot, the query is able to return a consistent view of the data to the client even as the underlying database is being modified by concurrent write operations.

Global Logical Clock

A foundational enhancement to MongoDB’s underlying architecture came with the introduction of a global logical clock and implementation of storage layer timestamps with the 3.6 release in 2017. The purpose of the clock and timestamps is to enforce consistent time across every operation in a distributed cluster. This enabled a host of new integrity and resilience capabilities including causal consistency, snapshot isolation, multi-document ACID...
transactions, and retryable writes – all of which are discussed in this paper.

Development work was based on an implementation of the earlier academic concepts of Lamport clocks and timestamps. A good resource to dig into the implementation of the logical clock is MongoDB's SIGMOD 2019 paper.

**Transactional Guarantees**

MongoDB added multi-document ACID transactions with the 4.0 release in 2018. You might ask how any database that didn't offer transactional guarantees across multiple records be used for anything more than very simple applications? The reality is that MongoDB has **always** offered transactional integrity through support for single document atomicity. For the vast majority of MongoDB applications single document atomicity provides the same data integrity guarantees as traditional relational databases.

It is able to do this because of the richness of MongoDB's document data model. Documents allow developers to collapse what would otherwise be separate parent-child tables in a relational database schema into a single JSON-like data structure. With this approach, one or more fields can be atomically written in a single operation, including updates to multiple subdocuments and elements of an array. Any errors on any of these write operations cause the entire write operation to roll back, and the document is isolated from any concurrent readers while the write is in-flight. As a result, clients receive a consistent view of the data.

Even with these guarantees, some developers and DBAs had been conditioned by 40 years of relational data modeling to assume multi-record transactions were a baseline requirement for any database, irrespective of the data model they are built upon. Some were concerned that while multi-document transactions were not initially needed by their apps, they could be in the future. And for some workloads, support for ACID transactions across multiple records was required.

It was for these reasons that MongoDB added support for multi-document ACID transactions in the 4.0 release and extended them to support distributed transactions across sharded clusters in 2019 with MongoDB 4.2. This allows developers to maintain the same transactional guarantees as single primary deployments – even when documents live in **different** partitions on **different** nodes.

For developers with a history of using transactions in relational databases, MongoDB’s multi-document transactions will be immediately familiar. They use a similar syntax and support snapshot isolation coupled with all-or-nothing execution. This makes it straightforward to add them to any application that needs them. To achieve the highest levels of transactional integrity, you must use the majority committed write concern and snapshot read concern, as discussed in the documentation.

**Schema Management and Versioning**

There are some who describe MongoDB (and “NoSQL” databases generally) as “schemaless”. Of course this isn’t true because every database has a schema of some sort, even if it’s implicitly defined in application code and by each document, rather than within the database itself.

What MongoDB does have is a **flexible schema**. This is a powerful feature that enables developers to store data of any structure, and to rapidly evolve their schema as new application features are developed.

However as applications reach steady-state after launch you might want more control over the structure of your data. Recognizing this requirement, MongoDB introduced Document Validation in 2015, and extended it with the 3.6 release in 2017 with Schema Validation, based on the IETF JSON Schema specification.

Using schema validation, you can define an explicit document structure for each collection, with the database rejecting or flagging any documents that do not conform to it. It can be used to enforce the presence of mandatory fields, specific data types and permissible values of each field, and optionally block the addition of new fields that have not been approved by the application owner. With schema validation, you have control to apply data governance against your document schema in production, while maintaining the benefits of a flexible and agile data model in development.
Whether you use schema validation or not, you can immediately view and explore your schema with MongoDB Compass and the Atlas Data Explorer (discussed later). This is really valuable when you inherit a MongoDB-based application and need to understand the data model. You can also control exactly how new attributes are rolled out across your collections – for example requiring all existing documents to be migrated to the new structure, or implementing changes "lazily" – only when the existing document is updated.

Database Resilience with Replica Sets and Elections

As noted earlier, a replica set is a group of MongoDB nodes that maintain the same data set. Replica sets are the basis for all production deployments of MongoDB. Principally they are designed to provide redundancy and high availability. They can also be used to provide low latency reads and the isolation of transactional and analytical workloads in the same physical database cluster.

Whenever you provision a cluster in MongoDB Atlas, a minimum of a 3-node replica set is instantiated with each replica automatically deployed into a separate availability zone in your chosen region. Because of this and the other resilience controls available in MongoDB, Atlas is able to offer a 99.995% uptime SLA guarantee.

Increasing resilience and read scalability, MongoDB 3.0, released in 2015, upped the maximum number of nodes in a single replica set from 12 to 50 members. Nodes in the replica set can be provisioned across data centers, geographic regions, and with MongoDB Atlas, fully-managed across different cloud providers, all with automatic failover and self-healing recovery.

MongoDB 3.2, released in late 2015, introduced a new replica set election protocol based on an extended implementation of the Raft consensus algorithm. By optimizing the algorithms used to detect node failures and elect a replacement primary, the new protocol delivered faster service recovery in the event of a primary failure. In MongoDB Atlas, failover and recovery is typically completed in five seconds or less. The Raft-based election protocol, combined with retryable reads and writes (discussed later), minimizes the impact to your application of both failures and planned maintenance.

Further reducing the impact of replica set elections continued to be major focus over the MongoDB 4.x releases:

- With MongoDB 4.2 in 2019, the stepDown process – used to take a primary down during maintenance and elect a replacement – was refactored to decrease the election time from a typical interval of 5 - 10 seconds to less than 1 second. The new stepDown process was also backported to MongoDB 3.6 and 4.0.
- MongoDB 4.4 in 2020 introduced Mirrored Reads. These are used to pre-warm the caches of electable replicas with the most recently accessed data, helping to rapidly restore performance after an election as the working set does not need to be loaded back from slower disk.

Improving the performance of replication itself, MongoDB 4.4 also introduced Streaming Replication. Rather than replicas polling the primary and receiving batches of events to apply locally, from 4.4 the primary continuously streams messages to the replicas. Testing has shown streaming replication reduces the latency of majority committed writes by up to 50% over high load networks, enabling developers to use the stricter write concern without introducing unnecessary latency to their applications.

Retryable Operations

Further enhancing resilience, MongoDB introduced Retryable Writes with the 3.6 release in 2017.

The addition of retryable writes moved the complexity of handling temporary system failures from the application to MongoDB. Rather than the developer having to implement custom client-side code, the MongoDB drivers automatically retry writes in the event of transient network failures or a primary replica election. With the MongoDB server enforcing exactly-once processing semantics, both idempotent and non-idempotent write operations can be automatically retried. Support for Retryable Reads was added to the drivers in the MongoDB 4.2 release in 2019.
Performance and Scalability

It goes without saying that for any application, the performance and scalability of its underlying database is a critical requirement. Developers need to be confident that their chosen database can deliver the SLAs demanded by the business from day one of production release and out into the future as the application grows.

Through both its underlying storage engine and sharding, MongoDB has made major improvements over the years to keep pace with developer demand and run their applications at scale.

WiredTiger Storage Engine

All early MongoDB releases through to 2014 used the MMAP (Memory Mapped) storage engine. While MMAP delivered a number of valuable features such as low latency reads and writes along with support for in-place document updates, its coarse-grained locking model limited write scalability. The MMAP storage engine originally enforced a global database write lock, with later releases moving to collection level locking. Developers could still scale write-heavy workloads, but it meant having to shard the workload earlier.

This all changed with the introduction of the WiredTiger storage engine in MongoDB 3.0, released in 2015. Developed by the original architects of Berkeley DB, WiredTiger offered new capabilities that transformed MongoDB performance and scalability:

- Fine-grained document level concurrency control, delivering a 7-10x out-of-the-box write performance improvement while maintaining 100% compatibility with existing MongoDB workloads built on MMAP.

- Native compression, reducing storage, journal, network, and index memory consumption by around 50% using the default snappy algorithm. Users could also opt for the more CPU-intensive zLib compression, reducing storage consumption by up to 80%.

- With MVCC support, WiredTiger also provided the storage layer foundations for multi-document transactions released later in MongoDB 4.0.

WiredTiger became MongoDB's default storage engine with the 3.2 release later in 2015 while MMAP was deprecated with MongoDB 4.0 in 2018. WiredTiger enhancements have continued over the years, including:

- Storage engine foundations for multi-document transaction support.

- Improved performance and scalability when handling large numbers of collections and indexes, capped collections, and small updates to large documents.

- The introduction of zStandard compression with MongoDB 4.2 in 2019. Originally developed at Facebook, zstd combines the high compression ratios of zLib with the low CPU overhead of snappy.

Sharding

Through native sharding, MongoDB horizontally scales-out your database across multiple nodes to handle read and write-intensive workloads and growing data sizes, without adding complexity to the application.

MongoDB's sharding has always been flexible, allowing you to optimize for different query patterns by controlling how data is distributed across your cluster. It does this by offering a choice of sharding schemes:

- Ranged-sharding co-locates data you regularly query together onto a single shard -- for example placing all orders for a given customer on one shard. This improves performance by minimizing "scatter-gather" cross-shard queries.

- Hashed-sharding applies an MD5 hash to the shard key value, providing an even distribution of data across shards at the expense of maintaining locality offered by ranged sharding.

MongoDB 3.4 released in 2016 introduced zoned-sharding, building on an earlier sharding scheme called tag-aware sharding. Zones allow developers to define policy-driven data placement in a sharded cluster. This accommodates advanced deployment scenarios -- for example pinning data to shards deployed in specific geographic regions for data sovereignty and low latency to local users. Zoned sharding is the foundation for MongoDB Atlas Global Clusters, allowing users to easily provision globally distributed clusters with shards located in different regions.
and even in different cloud providers, all through the Atlas GUI or its declarative API.

Further improving performance in sharded clusters, MongoDB 4.4 introduced Hedged Reads enabling the drivers to tell the mongos query router to submit read requests to multiple replicas, returning results to the client as soon as the quickest node responds. By hedging reads you can minimize p95 and p99 latencies in sharded clusters because queries avoid waiting on one node that might be busy syncing to disk, applying an index build, or experiencing a transient hardware issue.

Rebalancing Data in Sharded Clusters

High-scale database clusters are rarely static because the workloads they serve are typically highly dynamic. It is therefore important for clusters to be elastic so they can rebalance data across shards, and add and remove shards as applications evolve and scale.

MongoDB has made a number of enhancements over the years to make it easier to respond as data volumes grow and applications evolve:

• In MongoDB 3.4 shard balancing was significantly improved by adding support for parallel data migrations, allowing data rebalancing to complete up to 10x faster. Multiple node pairs were now able to perform balancing migrations simultaneously. In addition, balancer throttling that had been necessary with the older MMAP storage engine was eliminated.
• By concurrently fetching and applying documents as they were migrated between shards, balancing speeds were further improved by up to 40% with MongoDB 4.0 in 2018.

Even with the flexibility of MongoDB sharding, the inability to modify either the shard key or shard key value had been a frustration for some developers. This is because choosing a shard key upfront that could not be changed later to accommodate changing application requirements could lead to poor performance as data might end up not being evenly distributed across the cluster. To modify the shard key developers would either have to dump and reload the collection with a new shard key, or implement their own migration scripts. Both approaches incurred downtime, risk, and complexity.

Through developments in both MongoDB 4.2 released in 2019 and the 4.4 release 12-months later, sharding has become more flexible and adaptable:

• MongoDB 4.2 introduced the ability to modify shard key values to change the placement of a document in a cluster. This is useful when you want to move a document to a shard in a different geographic region.
• MongoDB 4.4 brought the ability to refine the shard key itself in a non-blocking operation, and to create compound hashed shard keys.

Through these capabilities, developers have more flexibility as their applications evolve and allow for an even finer-grained distribution of data within the cluster.

Tiered Scaling: Online Archive

Beyond vertical and horizontal scaling, MongoDB also offers tiered-scaling. When working in the cloud, MongoDB Atlas Online Archive will automatically tier infrequently accessed data out of the database onto fully managed cloud object storage.

Archived data remains fully accessible with federated queries that span both object and database storage in a single connection string. This approach enables you to more economically scale data storage by moving it to a lower cost storage tier without losing easy access to the data or grappling with slow and complex ETL pipelines.

Scaling Reads Across a Replica Set

By taking advantage of storage engine timestamps and snapshots, MongoDB 4.0 made it much easier to scale reads across a replica set. By reading from the latest snapshot prior to replication batches being applied to a replica, clients were now guaranteed a non-blocking and consistent view of the data, improving latency by up to 2x. Review the non-blocking secondary reads blog post to learn more.

Performance and Scalability in MongoDB Atlas

MongoDB Atlas offers the improvements made to performance and scalability over the years. It also provides
additional capabilities that make performance optimization and scaling even easier for developers:

- **Auto-scaling** that adjusts both compute and storage in response to application demand. By monitoring CPU and memory utilization, Atlas can automatically scale your database up and down. Atlas will also monitor storage consumption and automatically provision more disk space once consumed capacity hits 90%.

- The **Performance Advisor** monitors slow queries to provide automated index recommendations and suggestions to improve schema design.

- Through the Real Time Performance Panel, Query Profiler, and Custom Alerts you get **full performance visibility** into your Atlas database, enabling you to proactively remediate potential issues that, if left unchecked, could compromise application performance.

**Expansion of the MongoDB Query API and Drivers**

A valuable feature in relational databases is the richness of SQL, enabling developers to query data in many different ways. Through expansion of the MongoDB query API, drivers, and tooling, MongoDB has sought to offer this same versatility.

This expansion is designed to do two things:

1. **Increase the breadth** of applications developers can build with MongoDB.

2. **Improve the productivity of developers** by making it more natural and easier to work with data while also reducing the amount of application-side code they need to write.

**MongoDB Drivers**

The **MongoDB drivers and query API** are designed to be idiomatic to your programming language and tools, enabling you to work with **data as code**. With simple and concise calls to the database you can serve multiple classes of workload – from high throughput transactional applications to real-time analytics that search, transform, and aggregate complex data sets at scale.

The MongoDB drivers are developed in lockstep with the MongoDB database. This enables you to consume the latest features and enhancements discussed so far in this paper, along with the enhancements in the MongoDB query engine that we will discuss next.

In 2015 driver behavior was standardized across all of the programming languages supported by MongoDB. This standardization provided developers with a more consistent experience in whatever language they were writing code, while staying idiomatic to each language. The new specifications covered the CRUD and aggregation APIs, along with mechanisms for server discovery, selection, and monitoring. These specifications continue to define driver behavior today.

New MongoDB drivers have been added over the years to support new programming languages: the official **MongoDB Go driver** was released in 2018, and drivers for **Rust** and server-side **Swift** were added in 2020.

The MongoDB drivers have also evolved as asynchronous programming has become more popular. This started with **Motor async driver** for Python in 2013, with the C# and Java drivers following in 2015. Now most MongoDB drivers offer support for the asynchronous model.

**MongoDB Query API**

It was with the introduction of the **Aggregation Pipeline** back in 2012 that the MongoDB API expanded beyond CRUD operations and MapReduce. Modeled on the concept of Unix pipes, data flows through a series of composable aggregation pipeline stages where it is transformed and processed, allowing you to create sophisticated real-time analytics against your data in-place, without having to move it out to specialized analytics systems.

In the following section, we summarize key evolutions to the MongoDB API and query engine over the past few releases.

**MongoDB 3.2** released in December 2015:

- Gave the ability to join collections for richer analytics using the **$lookup aggregation pipeline stage**. $lookup offers left outer joins and was further extended in 2017
with support for uncorrelated subqueries between collections and more advanced join conditions beyond a single equality match.

- Improved database performance with the ability for the query optimizer to use covered indexes in aggregation pipelines.
- New **aggregation pipeline operators** for mathematical computations (i.e., ceiling, floor, absolute, rounding, square root, logarithms, standard deviations) and for manipulating elements of an array.

**MongoDB 3.4** released in November 2016:

- The addition of graph processing with the $graphLookup aggregation stage, used to rapidly traverse data sets to identify connections between them (i.e., friend-of-friend queries).
- Expansion of MongoDB’s BSON data format to include the **decimal data type**, allowing high-precision processing of complex numerical data such as monetary values (i.e., currency conversions) and scientific notations (i.e., genomic sequencing).
- Support for **collations**, enabling developers to create localized user experiences by specifying over 100 language-specific rules that govern sort order and string comparisons.
- Additional performance improvement by having the query optimizer automatically move the $match stage earlier in an aggregation pipeline to filter out unnecessary data from later processing stages. This was an important step to having the database optimize query performance, rather than push responsibility back to the developer. Other **aggregation enhancements** included new operators for string manipulation, array and type handling, and schema transformation.

**MongoDB 3.6** released in December 2017:

- **Change streams**, enabling developers to build reactive, real-time apps that can view and act on data changes as soon as they occur in the database. Implemented as an API on top of MongoDB’s oplog (used for replication in a MongoDB cluster) consuming applications can use change streams to subscribe to all changes in the database. When coupled with MongoDB Atlas Triggers, developers can more easily build reactive, event-driven apps that automatically execute server-side logic whenever a document is added, updated, or removed from the database – for example firing off a welcome email when a new customer subscribes to a service.
- Higher performance with fully expressive array updates using the **arrayFilters option**, allowing complex manipulations against matching elements of an array, all in a single atomic update operation. More expressivity was also added to the MongoDB API with the **$expr operator**, allowing developers to implement complex business logic in comparison queries.

**MongoDB 4.0** released in June 2018:

- Bringing ETL natively into the database, the **$convert operator** enabled the aggregation pipeline to transform inconsistent data types into standardized formats. Ingested data can be cast into a cleansed format (i.e., longs converted to decimals, strings to dates) before being exposed to consuming applications.
- Lowering the bar to building complex queries, MongoDB introduced the Aggregation Pipeline Builder, available via the Compass GUI (discussed later) and the Atlas Data Explorer. Developers can just drag and drop pipeline stages onto a canvas, visually constructing sophisticated pipelines in a low-code workflow. They can inspect the results at each aggregation stage to ensure correctness and optimize performance, and when done, export the pipeline created in the Compass GUI to native code for their application.

**MongoDB 4.2** released in August 2019:

- The addition of **On-Demand Materialized Views** gave developers the ability to precompute and cache the results of common analytics queries. Developers can increment and enrich the materialized view as new data is processed by the aggregation pipeline – for example adding new sales data for the most recent day’s trading – rather than force a complete reprocessing of the full sales data set.
- Updates were made faster and easier by using the
aggregation pipeline to create more expressive update statements. For example, developers could express conditional updates based on current field values or update one field using the value of other fields.

- Alongside 4.2, MongoDB Atlas Search was also released. Exposing a fully-managed Apache Lucene index directly on top of the database enabled developers to create rich, relevance-based search experiences without having to move their data into a separate search engine. Atlas Search offers advanced full-text search features including auto-complete, typo tolerance, and custom scoring – all fully integrated into the MongoDB API so developers don’t need to context switch to another query language.

MongoDB 4.4 released in July 2020:

- Through the `$unionWith` aggregation pipeline stage, developers can now blend data from multiple collections (for example, sales by month) into a single result set in the database, enabling deeper exploration and analysis of their data.
- Custom Aggregation Expressions, allow developers to extend the functionality of MongoDB for specific use cases by defining their own custom expressions in JavaScript and have them execute in the database as part of an aggregation pipeline.

Indexing

To efficiently support complex access patterns to your data, MongoDB has always offered a broad range of index types and properties, adding language-specific sort orders in 2015. MongoDB indexes can be declared on any set of fields within a document, including fields nested within arrays, and can be created and dropped on-demand to accommodate evolving application query patterns. Like the MongoDB query API and drivers, indexes have been extended over multiple releases.

Partial indexes introduced with MongoDB 3.2 only index documents that meet a specified filter expression (for example, only active customers who had placed an order in the past three months). By indexing a subset of the documents in a collection, partial indexes benefit from lower storage requirements and reduced performance costs for index creation and maintenance.

Wildcard indexes were introduced in MongoDB 4.2. Developers could now define a filter that automatically indexes all matching fields, subdocuments, and arrays in a collection. Wildcard indexes are especially valuable for ad-hoc queries and data discovery where query patterns are not known in advance. They also simplify schema design for polymorphic document structures typical in use-cases such as product catalogs and content management.

MongoDB 4.4 added hidden indexes, enabling a developer to hide an index from the query planner so they could evaluate the impact of removing it, without actually dropping the index. If they later determined the index was required, then they simply unhide it, avoiding the expense of a full index rebuild. The hidden index is fully maintained alongside all other indexes on the collection, and can be hidden and unhidden on-demand.

Beyond index types, the way in which indexes are created and managed has also been improved, allowing developers to more easily roll out indexes across their collections:

- MongoDB 4.2 introduced optimized online index builds that offer developers the best of the existing index build approaches in MongoDB: the speed of foreground builds coupled with the non-blocking behavior of indexes built as background processes.
- In MongoDB 4.4 indexing builds were changed so that they could be applied simultaneously across both primary and secondary replicas, reducing lag between replicas.

Tools and Connectors

While many developers want to interact programmatically with MongoDB through the language drivers or shell, there are also many who also want a GUI to work with the database. MongoDB Compass was first released back in 2015, and has continuously evolved since. Through the Compass GUI you can explore and manipulate your data, visually create queries and aggregation pipelines and then export them as code for your app; view and create indexes; import and export data; build schema validation rules; get real-time insight into query performance; and more.
In 2020, MongoDB introduced the MongoDB for VS Code extension. This allows developers to connect to their database and work with data directly within their VS Code environment. They can view their collections and schemas; search and edit documents; build queries and aggregations; as well as prototype and execute CRUD operations and other MongoDB commands – all as part of their regular workflow and tool chain.

For users wanting to visualize MongoDB data quickly and easily without the overhead of dedicated BI tools, MongoDB Charts – released in 2018 – provides a highly efficient solution. Charts can work directly with richly structured JSON documents in Atlas, avoiding the need to flatten schemas, and uses the MongoDB query API natively without requiring translation into SQL. Users can create graphs and build dashboards, sharing them with other teams for collaboration, and embed them directly into web apps to create engaging visualizations which are updated in real time as data in MongoDB changes.

To improve integration with a broader set of data consumers and applications, MongoDB has also created a suite of connectors:

- With the MongoDB Connector for BI released in 2015 you can use MongoDB as a data source for your existing SQL-based BI and analytics tools such as Tableau, Microstrategy, and Looker. Rather than ETL data out of MongoDB into a relational database the BI Connector creates a tabular representation of MongoDB’s document schema, and translates SQL statements into the MongoDB API. Results are then returned to the BI tool where they are visualized alongside regular SQL data sources.

- The MongoDB Connector for Apache Spark released in 2015 exposes all of Spark’s libraries, including Scala, Java, Python and R. MongoDB data is materialized as DataFrames and Datasets for analysis with machine learning, graph, streaming, and SQL APIs.

- The MongoDB Connector for Apache Kafka, released in 2019, makes it easier for developers to build robust data pipelines that move events between systems in real time. The connector enables MongoDB to be used both as a source and a sink for Kafka, and is integrated with MongoDB Change Streams discussed earlier, allowing events to be pushed straight into Kafka topics immediately after being written to the database. The connector is developed and supported by MongoDB and verified by Confluent.

Beyond the Backend Database: Mobile and Data Lake

While data is essential for any application, it needs to be managed in more than just a backend application database:

- With customers expecting mobile-first experiences, data needs to be managed directly on-device, out at the edge of the network, and synced to the backend.

- Developers are also increasingly tasked with building more contextualized, intelligent experiences for their users, requiring them to work with multiple data assets stored in a data lake.

MongoDB has expanded its platform with Realm and Atlas Data Lake to address these broader developer requirements:

- The MongoDB Realm Mobile Database enables you to store data locally on iOS and Android devices using a rich and intuitive data model, and accessed via idiomatic SDKs for Swift, Objective-C, Java, Kotlin, C#, and JavaScript. Optimized for mobile and edge platforms, Realm is lightweight, efficiently using memory, disk space, and battery life. As a local database, it persists data on-disk, so apps work as well offline as they do online.

- MongoDB Realm Sync, previewed in 2020 and released as GA in 2021, makes it simple to keep data in sync across users, devices, and the backend Atlas database in real-time, handling conflict resolution and networking code for you.

- With Realm Application Development Services, you can access the data stored in MongoDB Atlas directly from the client. The GraphQL service can be used with Realm Functions, Triggers, and Data Access Rules to simplify the code required to build secure and performant web apps. There is no need to manage back-end services as everything is delivered in a completely serverless model.

- With MongoDB Atlas Data Lake you can query and analyze data federated across cloud object storage and
MongoDB Atlas using the MongoDB API, delivered in a serverless query model. Spin up a data lake alongside your Atlas clusters using a common UI, and immediately start querying existing data assets in cloud object storage, including Parquet, Avro, Orc, JSON, BSON, CSV, and TSV files. You can run sophisticated aggregation pipelines to process and reshape data, persisting results back to your preferred storage tier.

Security and Privacy

Access control (authentication and authorization), encryption, and auditing are the pillars of data security needed to meet regulatory compliance mandates and preserve user privacy. MongoDB has always offered the requisite security controls, expanding them over multiple releases to meet the demands of an evolving cybersecurity and regulatory landscape. In this final section of the guide, we will step through each area to explore what’s new.

Network Access

Since the earliest releases of MongoDB, users have had the ability to restrict network access to the database. However developer responsibilities for securing the database have changed over time as MongoDB’s defaults have evolved:

- Prior to MongoDB 2.6 in 2014, all network connections to the database were open by default. This meant developers had to explicitly restrict access to MongoDB when it was first installed.
- From 2.6 onwards, the binaries from the official MongoDB RPM and DEB packages were hardened to bind only to localhost by default. This default behavior was extended in the 3.6 release to all MongoDB packages across all platforms. With these changed defaults, developers configure access only when MongoDB is opened up to users over the network. This change in defaults reduces the risk of an unsecured MongoDB instance being accidentally deployed. Note that MongoDB Atlas automatically enforces access control best practices. By default, your MongoDB Atlas database will have no access from the internet. Each Atlas database is deployed within a VPC, configured to allow no inbound access. You can then setup IP Access Lists to limit which IP addresses can attempt to authenticate to the database.

Access Control: Authentication

Once a MongoDB database is accessible on the network, administrators have multiple methods for user authentication, starting with password-based Salted Challenge Response Authentication Mechanism (SCRAM) authentication and then adding x.509 certificates in the 2.6 release. MongoDB 4.0 in 2018 updated SCRAM from SHA-1 to the stronger SHA-256 hashing mechanism.

In addition to user authentication, SCRAM and x.509 certificates can also be used to authenticate each node’s membership in a MongoDB cluster. The 4.2 release in 2019 added the ability for administrators to rotate the SCRAM keyfiles – used to store the nodes’ password credentials – without incurring application downtime.

Further simplifying cloud-native security specifically in Atlas, MongoDB 4.4 added support for Amazon IAM authentication. Your EC2 instances, containers, and serverless functions could now authenticate to MongoDB Atlas reusing existing Amazon IAM credentials, all managed through a single authentication mechanism.

Access Control: Authorization & Data Masking

Always-on and customizable authorization ensures that only authorized (and authenticated) users are allowed access to the database. Role-Based Access Control (RBAC) was added to MongoDB 2.4 back in 2013, allowing for a fine-grained separation of duties between different database users.

User privileges were further refined with the introduction of Read-Only Views in MongoDB 3.4. Views can be defined to expose only a subset of data from an underlying collection based on the user’s role – for example, filtering out or masking fields containing sensitive data, such as Personally Identifiable Information (PII). Read-only views are created each time the query is run by the client. As an alternative to read-only views, Materialized Views, added in MongoDB 4.2, allow the results of common queries to be cached, rather than having to be re-executed every time. Client-Side Field Level Encryption, discussed later, also provides a solution for masking sensitive data.
The 3.4 release also extended MongoDB's existing LDAP authentication to include authorization, enabling existing user privileges stored in a LDAP server to be mapped to MongoDB roles, without users having to be recreated in MongoDB itself. This reduces administrative overhead and allows easier MongoDB integration into a centralized enterprise directory infrastructure. LDAP is available in MongoDB Atlas and MongoDB Enterprise Advanced.

Encryption: Data In-Motion and At-Rest

MongoDB can automatically encrypt data in-motion over the network – between clients and the database, and between the nodes of a cluster. Ciphers have been upgraded over multiple releases as the industry has advanced cryptographic protocols:

- MongoDB 4.2 in 2019 added support for the latest TLS cipher suites offering Forward Secrecy, providing the assurance that your users’ session keys will remain secure even if the server’s private key is compromised.
- Starting with MongoDB 4.4 in 2020, support was added for TLS 1.3 when used with a compatible OpenSSL library. TLS 1.3 and the accompanying forward secret ciphers further help meet strict encryption in-motion requirements and were also backported to all supported MongoDB releases.

Note that MongoDB Atlas automatically enforces always-on TLS encryption.

Many users rely on volume encryption to protect data at-rest, and like in-motion encryption, this is always-on with MongoDB Atlas. While volume encryption prevents unauthorized users reading database records on the filesystem, it does not, however, protect against sysadmins or attackers with compromised credentials from reading persisted data in plaintext. To close this threat vector, MongoDB 3.2 introduced the Encrypted Storage Engine, allowing only those users and administrators with the appropriate database credentials to access encrypted data on disk and in backups.

Built on WiredTiger, the Encrypted storage engine is available as part of MongoDB Enterprise Advanced with KMIP integration, and as an option in MongoDB Atlas with customer key management, integrated with AWS, Azure, and Google Cloud key management systems.

Encryption: Data In-Use

While many databases offer the ability to encrypt data in-motion and at-rest, they lack the ability to natively encrypt data in-use, while it is in memory being processed by the database – leaving it vulnerable to attack.

For this reason, organizations handling highly sensitive data, especially in regulated industries, use application-level encryption to render data as ciphertext before it even hits the network and reaches the database. However this approach adds complexity to the application, and leaves the encrypted data as an opaque BLOB in the database, unable to be queried directly.

This changed with MongoDB Client-Side Field Level Encryption, introduced with MongoDB 4.2 and drivers in 2019. With Client-Side FLE your most sensitive data is automatically encrypted by the MongoDB drivers before leaving the application, and so the database only ever works with it as ciphertext. Whether resident in memory, in system logs, at-rest in storage, and in backups – all protected data remains unreadable. Developers do not need to write additional code; applications can still query data; and there is no significant impact to database performance.

By securing data with Client-Side FLE organizations have been able to do two things that could not be achieved with in-motion or at-rest encryption alone:

1. Move to managed services in the cloud with greater confidence. This is because even those who run the service or the underlying cloud infrastructure cannot decrypt the data.
2. Make it easier to comply with “right to erasure” mandates in modern privacy legislation. When a user invokes their right to erasure, you simply destroy the associated encryption key and the user’s Personally Identifiable Information (PII) is rendered unreadable and irrecoverable to anyone.

Security & Compliance in MongoDB Atlas

With MongoDB Atlas, your data is protected with preconfigured security features for authentication, authorization, encryption, and more.
Atlas undergoes independent verification of platform security, privacy, and compliance controls designed to help meet regulatory and policy objectives, such as HIPAA, PCI-DSS, GDPR, SOC 2 Type II, and ISO 27001. Review the MongoDB Trust Center to learn more. On that page you will also find a detailed whitepaper describing the security controls available in MongoDB Atlas.

Conclusion

MongoDB has evolved rapidly since its initial 1.0 release in 2009. In this paper, we’ve focused on key advancements in data integrity and resilience; performance and scalability; the versatility of the MongoDB query API and drivers; and security and privacy.

From MongoDB 5.0 onwards, the pace of database innovation will further accelerate as we move from an annual to a quarterly release cadence (and even higher frequency in the future). The new Versioned API will enable you to take advantage of the latest MongoDB releases and features without the fear of introducing incompatible changes that require application-side rework.

Today MongoDB counts over 150 million downloads, several million Atlas clusters under management, and 24,500 customers— including more than 50% of the Fortune 100— running applications on the database. The best way to evaluate whether MongoDB is suitable for your next application is to try it out on MongoDB Atlas. We’d also love to get your insight on future features and enhancements that would be most valuable to you via the MongoDB Feedback Engine.

Safe Harbor

The development, release, and timing of any features or functionality described for our products remains at our sole discretion. This information is merely intended to outline our general product direction and it should not be relied on in making a purchasing decision nor is this a commitment, promise or legal obligation to deliver any material, code, or functionality.

We Can Help

We are the company that builds and runs MongoDB. Over 24,500 organizations rely on our commercial products. We offer cloud services and software to make your life easier:

MongoDB Atlas is the global cloud database service for modern applications. Deploy fully managed MongoDB across AWS, Azure, or Google Cloud with best-in-class automation and proven practices that guarantee availability, scalability, and compliance with security standards.

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

MongoDB Atlas Data Lake allows you to quickly and easily query data in any format on Amazon S3 using the MongoDB Query Language and tools. You don’t have to move data anywhere, you can work with complex data immediately in its native form, and with its fully-managed, serverless architecture, you control costs and remove the operational burden.

MongoDB Charts is the best way to create, share and embed visualizations of MongoDB data. Build visualizations quickly and easily to analyze complex, nested data. Embed individual charts into any web application or assemble them into live dashboards for sharing.

Realm Mobile Database allows developers to store data locally on iOS and Android devices using a rich data model that’s intuitive to them. Combined with the MongoDB Realm sync-to-Atlas, Realm makes it simple to build reactive, reliable apps that work even when users are offline.

MongoDB Realm allows developers to validate and build key features quickly. Application development services like Realm Sync for mobile and Realm’s GraphQL service, can be used with Realm Functions, Triggers, and Data Access Rules— simplifying the code required to build secure and performant apps.

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](http://mongodb.com) or contact us at sales@mongodb.com.

- Case Studies ([mongodb.com/customers](http://mongodb.com/customers))
- Presentations ([mongodb.com/presentations](http://mongodb.com/presentations))
- Free Online Training ([university.mongodb.com](http://university.mongodb.com))
- Webinars and Events ([mongodb.com/events](http://mongodb.com/events))
- Documentation ([docs.mongodb.com](http://docs.mongodb.com))
- MongoDB Atlas database as a service for MongoDB ([mongodb.com/cloud](http://mongodb.com/cloud))
- MongoDB Enterprise Download ([mongodb.com/download](http://mongodb.com/download))
- MongoDB Realm ([mongodb.com/realm](http://mongodb.com/realm))