

MongoDB 4 Update: Multi-Document ACID Transactions

[Try MongoDB 4.0](#)

Mat Keep and Alyson Cabral

June 27, 2018 | Updated: June 27, 2020

[#Releases](#) [#Technical](#) [#MongoDB 4.0](#) [#Transactions](#)

With the release of 4.0, you now have multi-document ACID transactions in MongoDB.

But how have you been able to be so productive with MongoDB up until now? The database wasn't built overnight and many applications have been running mission critical use cases demanding the highest levels of data integrity without multi-document transactions for years. How was that possible?

Well, let's think first about why many people believe they need multi-document transactions. The first principle of relational data modeling is normalizing your data across tables. This means that many common database operations, like account creation, require atomic updates across many rows and columns.

In MongoDB, the data model is fundamentally different. The document model encourages users to store related documents together in a single document. MongoDB has always supported ACID transactions in a single document and, when leveraging the document model appropriately, many applications don't need ACID guarantees across multiple documents.

However, transactions are not just a check box. Transactions, like every MongoDB feature, aim to make developers' lives easier. ACID guarantees across documents simplify application logic needed to satisfy complex applications.

The value of MongoDB's transactions

While MongoDB's existing atomic single-document operations already provide transaction semantics that satisfy the majority of applications, the addition of multi-document ACID transactions makes it easier than ever for developers to address the full spectrum of use cases with MongoDB. Through snapshot isolation, transactions provide a consistent view of data and enforce all-or-nothing execution to maintain data integrity. For developers with a history of transactions in relational databases, MongoDB's multi-document transactions are very familiar, making it straightforward to add them to any application that requires them.

In MongoDB 4.0, transactions work across a replica set, and MongoDB 4.2 will extend support to transactions across a sharded deployment* (see the Safe Harbor statement at the end of this blog). Our path to transactions represents a multi-year engineering effort, beginning back in early 2015 with the groundwork laid in almost every part of the server and the drivers. We are feature complete in bringing multi-document transactions to a replica set, and 90% done on implementing the remaining features needed to deliver transactions across a sharded cluster.

In this blog, we will explore why MongoDB had added multi-document ACID transactions, their design goals and implementation, characteristics of transactions and best practices for developers. You can get started with MongoDB 4.0 now by spinning up your own fully managed, on-demand [MongoDB Atlas cluster](#), or [downloading it](#) to run on your own infrastructure.

Why Multi-Document ACID Transactions

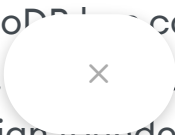
Since its first release in 2009, MongoDB has continuously innovated around a new approach to database design,  developers from the constraints of legacy relational databases. A design founded on rich, natural, and flexible documents accessed by idiomatic programming language APIs, enabling developers to build apps 3-5x faster. And a distributed systems architecture to handle more data, place it where users need it, and maintain always-on availability. This approach has enabled developers to create powerful and sophisticated applications in all industries, across a tremendously wide range of use cases.



Figure 1: Organizations innovating with MongoDB

With subdocuments and arrays, documents allow related data to be modeled in a single, rich and natural data structure, rather than spread across separate, related tables composed of flat rows and columns. As a result, MongoDB's existing single document atomicity guarantees can meet the data integrity needs of most applications. In fact, when leveraging the richness and power of the document model, we estimate 80%-90% of applications don't need multi-document transactions at all.

However, some developers and DBAs have been conditioned by 40 years of relational data modeling to assume multi-document transactions are a requirement for any database, irrespective of the data model they are built upon. Some are concerned that while multi-document transactions aren't needed by their apps today, they might be in the future. And for some workloads, support for ACID transactions across multiple documents is required.

As a result, the addition of multi-document transactions makes it easier than ever for developers to address a complete range of use cases on MongoDB. For

some, simply knowing that they are on a flexible platform will assure them that they can evolve their application as needed, and a database will support them.

Data Models and Transactions

Before looking at multi-document transactions in MongoDB, we want to explore why the data model used by a database impacts the scope of a transaction.

Relational Data Model

Relational databases model an entity's data across multiple records and parent-child tables, and so a transaction needs to be scoped to span those records and tables. The example in Figure 2 shows a contact in our customer database, modeled in a relational schema. Data is normalized across multiple tables: customer, address, city, country, phone number, topics and associated interests.

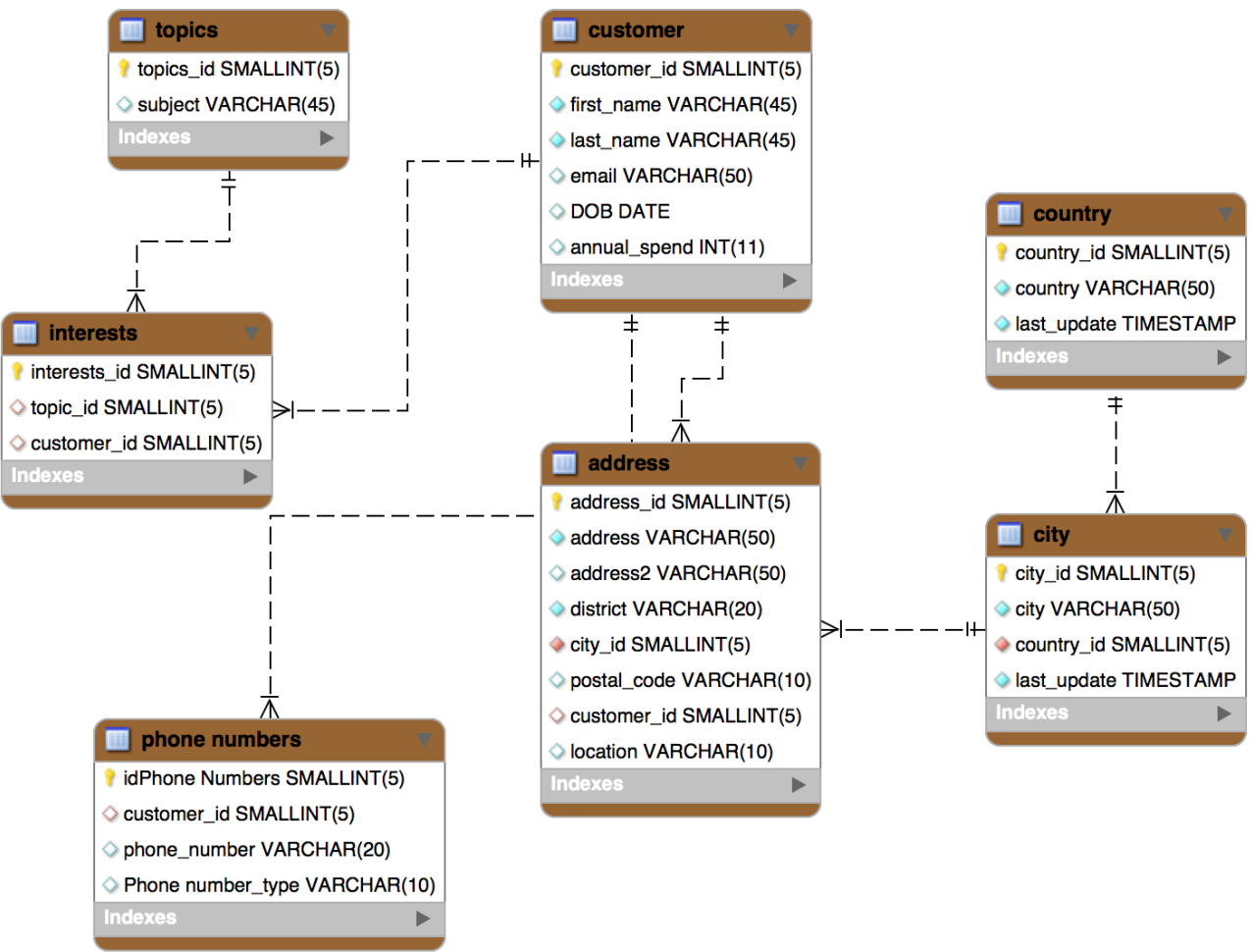


Figure 2: Customer data modeled across separate tables in a relational database

In the event of the customer data changing in any way, for example if our contact moves to a new job, then multiple tables will need to be updated in an

“all-or-nothing” transaction that has to touch multiple tables, as illustrated in Figure 3.

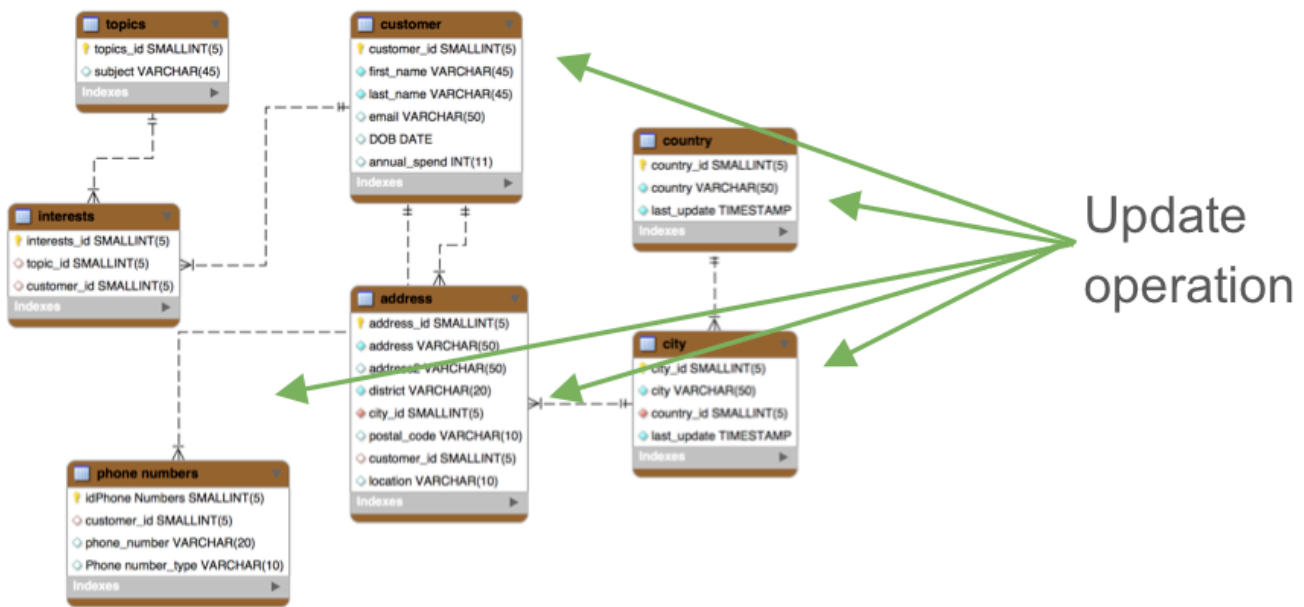


Figure 3: Updating customer data in a relational database

Document Data Model

Document databases are different. Rather than break related data apart and spread it across multiple parent-child tables, documents can store related data together in a rich, typed, hierarchical structure, including subdocuments and arrays, as illustrated in Figure 4.

```
_id: 12345678
> name: Object
> address: Array
> phone: Array
  email: "john.doe@mongodb.com"
  dob: 1966-07-30 01:00:00.000
  interests: Array
    0: "Cycling"
    1: "IoT"
```

Figure 4: Customer data modeled in a single, rich document structure

MongoDB provides existing transactional properties scoped to the level of a document. As shown in Figure 5, one or more fields may be written in a single operation, with updates to multiple subdocuments and elements of any array, including nested arrays. The guarantees provided by MongoDB ensure complete

isolation as a document is updated; errors cause the operation to roll back so that clients receive a consistent document. With this design, application owners get the same data integrity guarantees as those provided by older relational databases.

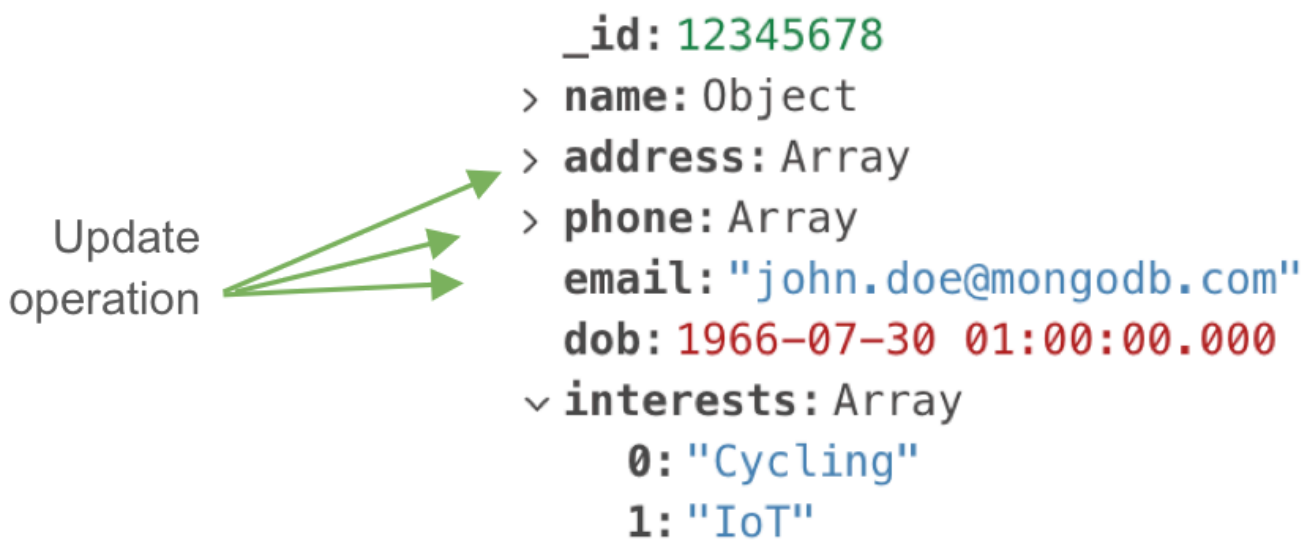


Figure 5: Updating customer data in a document database

Where are Multi-Document Transactions Useful

There are use cases where transactional ACID guarantees need to be applied to a set of operations that span multiple documents. Back office “System of Record” or “Line of Business” (LoB) applications are the typical class of workload where multi-document transactions are useful. Examples include:

- Processing application events when users perform important actions -- for instance when updating the status of an account, say to delinquent, across all those users’ documents.
- Logging custom application actions -- say when a user transfers ownership of an entity, the write should not be successful if the logging isn’t.
- Many to many relationships where the data naturally fits into defined objects -- for example positions, calculated by an aggregate of hundreds of thousands of trades, need to be updated every time trades are added or modified.

MongoDB already serves these use cases today, but the introduction of multi-document transactions makes it easier as the database automatically handles multi-document transactions for you. Before their availability, the developer would need to programmatically implement transaction controls in their application. To ensure data integrity, they would need to ensure that all stages of the operation succeed before committing updates to the database, and if not, roll back any changes. This adds complexity that slows down the rate of application development. MongoDB customers in the financial services industry

have reported they were able to cut 1,000 lines of code from their apps by using multi-document transactions.

×

In addition, implementing client side transactions can impose performance overhead on the application. For example, after moving from its existing client-side transactional logic to multi-document transactions, a global enterprise data management and integration ISV experienced improved MongoDB performance in its Master Data Management solution: throughput increased by 90%, and latency was reduced by over 60% for transactions that performed six updates across two collections.

Multi-Document ACID Transactions in MongoDB

Transactions in MongoDB feel just like transactions developers are used to in relational databases. They are multi-statement, with similar syntax, making them familiar to anyone with prior transaction experience.

The following Python code snippet shows a sample of the transactions API.

```
with client.start_session() as s:
    s.start_transaction()
    collection_one.insert_one(doc_one, session=s)
    collection_two.insert_one(doc_two, session=s)
    s.commit_transaction()
```

The following snippet shows the transactions API for Java.

```
try (ClientSession clientSession = client.startSession()) {
    clientSession.startTransaction();
    collection.insertOne(clientSession, docOne);
    collection.insertOne(clientSession, docTwo);
    clientSession.commitTransaction();
}
```

As these examples show, transactions use regular MongoDB query language syntax, and are implemented consistently whether the transaction is executed across documents and collections in a replica set, and with MongoDB 4.2, across a sharded cluster*.

“ We're excited to see MongoDB offer dedicated support for ACID transactions in their data platform and that our collaboration is manifest in the Lovelace release of Spring Data MongoDB. It ships with the well known Spring annotation-driven, synchronous transaction support using the `MongoTransactionManager` but also bits for reactive transactions built on top of MongoDB's `ReactiveStreams` driver and Project Reactor datatypes exposed via the `ReactiveMongoTemplate`.

”

Pieter Humphrey - Spring Product Marketing Lead, Pivotal

The transaction block code snippets below compare the MongoDB syntax with that used by MySQL. It shows how multi-document transactions feel familiar to anyone who has used traditional relational databases in the past.

MySQL

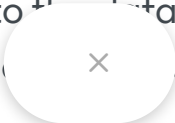
```
db.start_transaction()  
    cursor.execute(orderInsert, orderData)  
    cursor.execute(stockUpdate, stockData)  
db.commit()
```

MongoDB

```
s.start_transaction()  
    orders.insert_one(order, session=s)  
    stock.update_one(item, stockUpdate, session=s)  
s.commit_transaction()
```

Through snapshot isolation, transactions provide a consistent view of data, and enforce all-or-nothing execution to maintain data integrity. Transactions can apply to operations against multiple documents contained in one, or in many, collections and databases. The changes to MongoDB that enable multi-document transactions do not impact performance for workloads that don't require them.

During its execution, a transaction is able to read its own uncommitted writes, but none of its uncommitted writes will be seen by other operations outside of the transaction. Uncommitted writes are not replicated to secondary nodes

until the transaction is committed to the database. Once the transaction has been committed, it is replicated and  atomically to all secondary replicas.

An application specifies write concern in the transaction options to state how many nodes should commit the changes before the server acknowledges the success to the client. All uncommitted writes live on the primary exclusively.

Taking advantage of the transactions infrastructure introduced in MongoDB 4.0, the new **snapshot read concern** ensures queries and aggregations executed within a read-only transaction will operate against a single, isolated snapshot on the primary replica. As a result, a consistent view of the data is returned to the client, irrespective of whether that data is being simultaneously modified by concurrent operations. Snapshot reads are especially useful for operations that return data in batches with the `getMore` command.

Even before MongoDB 4.0, typical MongoDB queries leveraged WiredTiger snapshots. The distinction between typical MongoDB queries and snapshot reads in transactions is that snapshot reads use the same snapshot throughout the duration of the query. Whereas typical MongoDB queries may switch to a more current snapshot during yield points.

Snapshot Isolation and Write Conflicts

When modifying a document, a transaction locks the document from additional changes until the transaction completes. If a transaction is unable to obtain a lock on a document it is attempting to modify, likely because another transaction is already holding the lock, it will immediately abort after 5ms with a write conflict.

However, if a typical MongoDB write outside of a multi-document transaction tries to modify a document that is currently being held by a multi-document transaction, that write will block behind the transaction completing. The typical MongoDB write will be infinitely retried with backoff logic until **\$maxTimeMS** is reached. Even prior to 4.0, all writes were represented as WiredTiger transactions at the storage layer and it was possible to get write conflicts. This same logic was implemented to avoid users having to react to write conflicts client-side.

Reads do not require the same locks that document modifications do.

Documents that have uncommitted changes from a transaction can still be read by other operations, and of course those operations will only see the committed values, and not the uncommitted state.

Only writes trigger write conflicts within MongoDB. Reads, in line with the expected behavior of snapshot isolation, do not prevent a document from being modified by other operations. Changes will not be surfaced as write conflicts unless a write is performed on the document. Additionally, no-op updates – like setting a field to the same value that it already had – can be optimized away before reaching the storage engine, thus not triggering a write conflict. To guarantee that a write conflict is detected, perform an operation like incrementing a counter.

Retrying Transactions

MongoDB 4.0 introduces the concept of error labels. The transient transaction error label notifies listening applications that the error surfaced, ranging from network errors to write conflicts, that the error may be temporary, and that the transaction is safe to retry from the beginning. Permanent errors, like parsing errors, will not have the transient transaction error label, as rerunning the transaction will never result in a successful commit.

One of the core values of MongoDB is its highly available architecture. These error labels make it easy for applications to be resilient in cases of network blips or node failures, enabling cross document transactional guarantees without sacrificing use cases that must be always-on.

Transactions Best Practices

As noted earlier, MongoDB's single document atomicity guarantees will meet 80-90% of an application's transactional needs. They remain the recommended way of enforcing your app's data integrity requirements. For those operations that do require multi-document transactions, there are several best practices that developers should observe.

Creating long running transactions, or attempting to perform an excessive number of operations in a single ACID transaction can result in high pressure on WiredTiger's cache. This is because the cache must maintain state for all subsequent writes since the oldest snapshot was created. As a transaction uses

the same snapshot while it is running. Writes accumulate in the cache throughout the duration of the transaction. These writes cannot be flushed until transactions currently running on old snapshots commit or abort, at which time the transactions release their locks and WiredTiger can evict the snapshot. To maintain predictable levels of database performance, developers should therefore consider the following:

1. By default, MongoDB will automatically abort any multi-document transaction that runs for more than 60 seconds. Note that if write volumes to the server are low, you have the flexibility to tune your transactions for a longer execution time. To address timeouts, the transaction should be broken into smaller parts that allow execution within the configured time limit. You should also ensure your query patterns are properly optimized with the appropriate index coverage to allow fast data access within the transaction.
2. There are no hard limits to the number of documents that can be read within a transaction. As a best practice, no more than 1,000 documents should be modified within a transaction. For operations that need to modify more than 1,000 documents, developers should break the transaction into separate parts that process documents in batches.
3. In MongoDB 4.0, a transaction is represented in a single oplog entry, therefore must be within the 16MB document size limit. While an update operation only stores the deltas of the update (i.e., what has changed), an insert will store the entire document. As a result, the combination of oplog descriptions for all statements in the transaction must be less than 16MB. If this limit is exceeded, the transaction will be aborted and fully rolled back. The transaction should therefore be decomposed into a smaller set of operations that can be represented in 16MB or less.
4. When a transaction aborts, an exception is returned to the driver and the transaction is fully rolled back. Developers should add application logic that can catch and retry a transaction that aborts due to temporary exceptions, such as a transient network failure or a primary replica election. With [retryable writes](#), the MongoDB drivers will automatically retry the commit statement of the transaction.
5. DDL operations, like creating an index or dropping a database, block behind active running transactions on the namespace. All transactions that try to newly access the namespace while DDL operations are pending, will not be able to obtain locks, aborting the new transactions.

You can review all best practices in the [MongoDB documentation for multi-document transactions](#).

Transactions and Their Impact to Data Modeling in MongoDB

Adding transactions does not make MongoDB a relational database – many developers have already experienced that the document model is superior to the relational one today.

All best practices relating to MongoDB data modeling continue to apply when using features such as multi-document transactions, or fully expressive

JOINS (via the [\\$lookup aggregation pipeline stage](#)). Where practical, all data relating to an entity should be stored in a single, rich document structure. Just moving data structured for relational tables into MongoDB will not allow users to take advantage of MongoDB's natural, fast, and flexible document model, or its distributed systems architecture.

The [RDBMS to MongoDB Migration Guide](#) describes the best practices for moving an application from a relational database to MongoDB.

The Path to Transactions

Our path to transactions represents a multi-year engineering effort, beginning over 3 years ago with the integration of the WiredTiger storage engine. We've laid the groundwork in practically every part of the platform – from the storage layer itself to the replication consensus protocol, to the sharding architecture. We've built out fine-grained consistency and durability guarantees, introduced a global logical clock, refactored cluster metadata management, and more. And we've exposed all of these enhancements through APIs that are fully consumable by our drivers. We are feature complete in bringing multi-document transactions to a replica set, and 90% done on implementing the remaining features needed to deliver transactions across a sharded cluster.

Figure 6 presents a timeline of the key engineering projects that have enabled multi-document transactions in MongoDB, with status shown as of June 2018. The key design goal underlying all of these projects is that their implementation does not sacrifice the key benefits of MongoDB – the power of the document model and the advantages of distributed systems, while imposing no performance impact to workloads that don't require multi-document transactions.

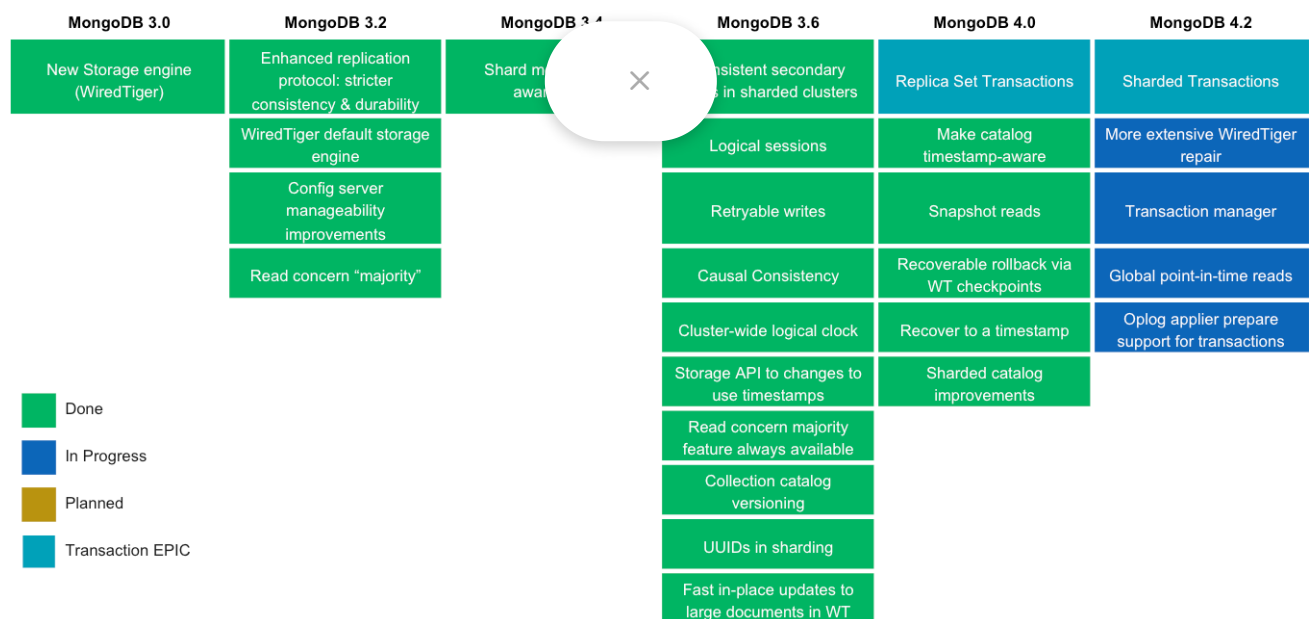


Figure 6: The path to transactions – multi-year engineering investment, delivered across multiple releases

Conclusion

MongoDB has already established itself as the leading database for modern applications. The document data model is rich, natural, and flexible, with documents accessed by idiomatic drivers, enabling developers to build apps 3-5x faster. Its distributed systems architecture enables you to handle more data, place it where users need it, and maintain always-on availability. MongoDB's existing atomic single-document operations provide transaction semantics that meet the data integrity needs of the majority of applications. The addition of multi-document ACID transactions in MongoDB 4.0 makes it easier than ever for developers to address a complete range of use cases, while for many, simply knowing that they are available will provide critical peace of mind.

Take a look at our [multi-document transactions web page](#) where you can hear directly from the MongoDB engineers who have built transactions, review code snippets, and access key resources to get started. You can get started with MongoDB 4.0 now by spinning up your own fully managed, on-demand [MongoDB Atlas cluster](#), or [downloading it](#) to run on your own infrastructure.

If you want to learn more about everything that's new in MongoDB 4.0, [download our Guide](#).

“ Transactional guarantees have been a critical feature for relational databases for decades, but have typically been absent from non-relational alternatives, which has meant that users have been forced to choose between transactions and the flexibility and versatility that non-relational databases offer. With its support for multi-document ACID transactions, MongoDB is built for customers that want to have their cake and eat it too.

”

Stephen O’Grady, Principal Analyst, Redmonk

*Safe Harbour Statement

This blog contains “forward-looking statements” within the meaning of Section 27A of the Securities Act of 1933, as amended, and Section 21E of the Securities Exchange Act of 1934, as amended. Such forward-looking statements are subject to a number of risks, uncertainties, assumptions and other factors that could cause actual results and the timing of certain events to differ materially from future results expressed or implied by the forward-looking statements. Factors that could cause or contribute to such differences include, but are not limited to, those identified our filings with the Securities and Exchange Commission. You should not rely upon forward-looking statements as predictions of future events. Furthermore, such forward-looking statements speak only as of the date of this presentation.

In particular, the development, release, and timing of any features or functionality described for MongoDB products remains at MongoDB’s 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. Except as required by law, we undertake no obligation to update any forward-looking statements to reflect events or circumstances after the date of such statements.



[← Previous](#)



MongoDB Charts Beta Now Available

Today at MongoDB World 2018 in New York, we are excited to announce the beta release of MongoDB Charts,...

June 27, 2018

[Next →](#)

MongoDB AI Applications Program Partner Spotlight: Cohere Brings Leading AI...

Today, Cohere, a leading enterprise AI platform, will join MongoDB's new AI Applications Program (MAAP) as...

May 1, 2024



About

Careers

Investor Relations

Legal Notices

Privacy Notices

Security Information

Trust Center

Support

Contact Us

Customer Portal

Atlas Status

Customer Support

Manage Cookies

Social

 GitHub

 Stack Overflow

 LinkedIn

 YouTube

 Twitter

 Twitch

 Facebook