MongoDB Data Versioning Performance: local versus Atlas

Lucia de Espona[1,†], Ela Pustulka[1,*,†]

[1]School of Business, University of Applied Sciences and Arts Northwestern Switzerland FHNW, Riggenbachstrasse 16, 4600 Olten, Switzerland

Abstract

We focus on versioning for NoSQL data on MongoDB. Versioning is essential for security audits, legal compliance and business strategy development. For each type of business object, we maintain two collections and split the data into currently valid objects and the archive. We previously presented a versioning algorithm and a preliminary evaluation on a local database, with version write times of 4 to 13ms, and versioning queries in 2 to 4ms. Here, we introduce bulk operations and present a performance study measuring all versioning operations on MongoDB Atlas. Cloud experiments show that version writes take 14 to 60ms. Queries need between 14 and 25ms. Using bulk versioning is faster, as writes need between 3 and 10ms. Cloud times are slower than those seen in local tests but the performance penalty due to network latency, imposed by the use of Atlas, is mitigated by the newly added bulk operations. Overall, the experiments show that the performance is satisfactory for an enterprise resource planning (ERP) system for small and medium enterprises (SMEs).

Keywords

database, NoSQL, document versioning, CRUD, ERP, MongoDB, performance, bulk operations, cloud

1. Introduction

We are working on an ERP system for an SME which has to comply with the legal requirement of keeping old data for ten years. This requirement has led us to explore versioning. Adding versioning to a business system enables queries that typically belong to the data warehousing scenario and support business in reporting and strategy development. When a legal requirement has to be fulfilled, the data can be queried flexibly.

Data versioning and archival are important aspects of database (DB) operations. Bernstein and Goodman [1] define a multi-version DB as one where each write on a data item x produces a new copy (or version) of x and for each read on x, the DB management system (DBMS) selects one of the versions of x to be read. Stonebraker and Rowe [2] outlined three types of versioning: *no archive* where no historical access to a relation is needed, *light archive* where archival is needed but this data will rarely be accessed, and *heavy archive* when the system needs to look up and update timestamps of previous transactions. In an ERP, we expect to see *no archive* for business objects that do not change and *light archive* for compliance, security and strategy queries. We do not see a need for a *heavy archive*.

Our ERP has versioning as a primary business requirement. Part of the data, such as customer or product details, will change, which is why versioning is required. Some data will be more stable and will only rarely change or not at all. As our business partner selected MongoDB as the platform, we decided to add versioning directly instead of using another tool as an archive, which would make business intelligence hard to orchestrate. Although some DBMSs offer built-in data versioning, in most cases, the versioning support is not sufficient for compliance [3] and data warehousing solutions are used [4, 5]. Mainstream NoSQL DBMSs, including MongoDB [6], only provide limited support for data versioning [7].

Here, we extend our previous work [8] where we presented a versioning algorithm which ensures that the query time on the currently valid document versions is kept constant, by storing historical data in a separate collection.

Earlier steps in this development were presented in [9] where we showed that the ERP fits the business use case. Our contributions here are: (1) performance tests on MongoDB Atlas cluster executing single document versioning and comparison to local performance we presented in [8], (2) a bulk versioning algorithm which extends the range of DB operations for MongoDB and uses transactions to support bulk requests, and (3) performance measurement of bulk INSERT, UPDATE, and DELETE on MongoDB Atlas and comparison with single document versioning.

2. Related work

Archival storage systems usually do not support flexible querying while software versioning systems offer queries based on taxonomies or text, but support no query language. Version control in most collaborative tools does not offer support for business data analysis or legal compliance.

In relational DBs, schema evolution and versioning for

DataPlat'23: 2nd International Workshop on Data Platform Design, Management, and Optimization, March 28, 2023, Ioannina, Greece

lucia.espona@gmail.com (L. d. Espona); elzbieta.pustulka@fhnw.ch (E. Pustulka)
0000-0002-1477-6999 (L. d. Espona); 0000-0001-7379-847X (E. Pustulka)

relations have been researched extensively [1, 2, 10, 11] and produced several interesting systems like ODM Insights [12]. Most NoSQL work focuses on schema changes [13, 14] and not on the data itself.

Here, we study data versioning and not schema versioning. Despite advances in data versioning, none of the prior systems fits our use scenario. The only publicly known document versioning library for MongoDB is Vermongo [15, 16]. The core idea we adopt from Vermongo is to store the current and past document versions separately as only the current data are of business interest on a daily basis. However, Vermongo has the following deficiencies which make it unsuitable for ERP: no ACID guarantees, no support for DBRefs which correspond to foreign keys in a relational DB and no support for multiple CRUD operations defined by Mongoose, such as *updateOne* and *updateMany*.

3. Versioning

We summarize the versioning solution we presented in [8]. In the MongoDB document versioning pattern [17], a version number is added to each document and the DB contains two collections for each original collection under version control: one holding the latest version and another one that holds all previous versions. The pattern assumes that most of the queries refer to the current document version and the historical data are accessed rarely, similarly to [18].

When a new business object type is added to the system, two collections are created: a main collection and a shadow collection. The new object is added to the *Main Collection*. When an object is edited, its old versions are placed in the *Shadow Collection* and the main collection stores only the current version. In the shadow collection, the *(objectId, version)* combination is unique but the original object ids are not unique and any other uniqueness constraints are automatically removed, to support the storage of multiple old versions of the same document, sharing some unchanged field values. In the shadow collection two indexes are defined by default: one on the unique object id (combination of the main collection object id and version number), and the other on the original id from the main collection and the validity timestamps. The versioning algorithm works as follows.

1. On *create*, a new document is written to the main collection.
2. On *update*, a new version is written to the main collection and the previous version is moved to the shadow collection, within a transaction.
3. On *delete*, the invalid version is moved to the shadow collection, within a transaction.
4. On *read*, both main and shadow collections can be queried.

Our solution relies on Mongoose [19]. Once the document model has been defined using Mongoose, the versioning plugin is added. It generates the two collections and adds the versioning related fields. Our implementation is available at [20, 21]. The plugin offers two query methods: *findVersion (id, version)* and *findValidVersion (id, date)* based on validity date.

Transactions are used in updates and deletes to guarantee operation atomicity, as they affect both the main and shadow collections. In contrast to Vermongo, which does not offer ACID guarantees, our implementation uses MongoDB transactions. Transactions are not needed for inserts which only affect the main collection. Transactions are handled by the user and passed to the versioning library for each enveloped operation.

4. Bulk Operations

Bulk operations are one of our three contributions. They optimize performance on the MongoDB Atlas cloud, since they reduce the network latency overhead by reducing the number of DB calls. MongoDB supports insert, update, and delete operations in bulk. Bulk writes on one collection are executed through the collection method *bulkWrite*. Bulk writes are also available in Mongoose through the Model API as *bulkWrite*, *insertMany*, *updateMany* and *deleteMany* in a single DB call. These operations are submitted as *bulkWrite* calls to the MongoDB driver.

We implement bulk versioning as: *bulkSaveVersioned*, a bulk insert or update equivalent to Mongoose *insertMany* and *updateMany*, and *bulkDeleteVersioned*, a bulk delete equivalent to Mongoose *deleteMany*. We had to implement new operations as the previous document versions have to be provided as input to guarantee the bulk operation performance gain. If we simply used the middleware hooks from Mongoose, the bulk operation would receive only the new version of the documents and it would need to query the DB for each object to obtain the past version, which would worsen performance.

Similar to single writes, bulk writes need to be wrapped inside a transaction if they operate over two collections to ensure consistency. Calling *bulkSaveVersioned* with no past versions provided is translated into a single bulk insert into the main collection and does not need a transaction. If we provide a set of past versions together with the new versions to perform an update using *bulkSaveVersioned*, this translates into a bulk insert into the main collection plus a bulk insert of the past versions into the shadow collection. This call is wrapped inside a transaction as it operates over two collections. *BulkDeleteVersioned* always needs to be called inside a transaction since it performs a bulk delete in the main collection and a bulk insert on the shadow collection.

5. Evaluation

The evaluation scenario mimics a use case of a human resources (HR) ERP system and is carried out on two types of infrastructure: a local machine, as in our previous paper [8] and in the cloud (MongoDB Atlas cluster). We compare the performance of three approaches: *no versioning*, *plain versioning* without using a shadow collection (all data, current and old in one collection), and *our versioning* which uses two collections, as outlined previously. HR data have an average size of 2.3K per document holding complex nested documents and document arrays storing personal data, skills, projects and similar.

The local infrastructure is described in our previous paper [8]. The cloud tests were submitted from a Linux server cluster (8GB RAM, 2 virtual CPUs and 20GB Disk with Ubuntu Jammy 22.04) which invoked DB operations on an M10 Atlas cluster with MongoDB version 4.2.17 Enterprise, a replica set composed of three nodes with the default DB configuration. Some system background tasks may affect results but should be similar for all tests. Performance was measured at the application level, via a Mongoose API call, including a transaction where needed, since the software introduces an overhead inherent to the solution design that needs to be reported, as reporting the times from MongoDB directly does not correspond to our use case scenario. The experiments can be reproduced using the code available at [22].

We tested the following operations:

- INSERT a new document
- UPDATE an existing document
- DELETE an existing document
- FVaNOW: find the current version by object id
- FVaPAST: find a past version by id and date
- FVe2: find current version by id and version no.
- FVe1: find past version by id and version no.
- FIND: find by non-indexed field on currently valid documents.

The operations were executed in groups of 100K in the local experiment as in [8] and 10K for the cloud. The order in the list represents the execution order inside each group of 100K (or 10K) and the group size corresponds to the maximum number of valid documents at any time point, corresponding to an HR system for a company with a constant employee count.

In the *local* experiment reported in [8], one million operations of each type were executed in groups of 100K lasting approximately 60 hours. There were 100K valid documents at any time.

As the *cloud* evaluation which aimed to replicate the local experiment took much longer per operation, we reduced the number of operations of each type to 100K and executed those in groups of 10K, with 10K valid documents at any time. The experiment took approximately 72 hours.

We carried out a *second cloud experiment to test bulk operations*. Here, INSERT, UPDATE, and DELETE were replaced with their bulk versions called InsertMany, UpdateMany, and DeleteMany. Each operation was performed on a group of documents sent as a single data chunk to the cloud, using two chunk sizes of 50 and 200 operations.

5.1. Local Single Operations versus Cloud

Performance on a local DB is detailed in [8]. The cloud experiment repeats the local experiment on the MongoDB Atlas cluster. The experiment size was reduced from 1M to 100K operations of each type (Insert, Update, Delete, FVaNow, FVaPast, FVe1 and FVe2) and the group size from 100K to 10K. Figure 1 compares local versioning performance to cloud performance.

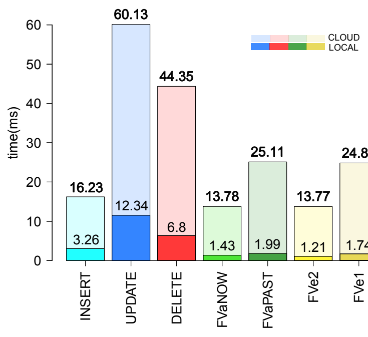

Figure 1: Local and cloud versioning performance. Execution times (ms) for the insert, update, delete and find, averaged over 1M executions locally and over 100K in the cloud. Update is the slowest, followed by delete, while insert and read operations perform similarly. Both update and delete use a transaction spanning two collections.

In both experiments, local and cloud, updates and deletes are the slowest, as they involve writes in both the main and the shadow collection wrapped in a transaction spanning both collections. The times are adequate for an SME ERP and even the slowest operation, the Update, needs 12-13 ms on average locally and 60 ms in the cloud, which is acceptable [23].

Figure 2 shows the details of the cloud measurements. Updates and deletes are the slowest, and the relative performance of all the operations is similar to the local performance presented in [8], except that all the single cloud operations are significantly slower in the Atlas cluster, which we interpret as network latency effect. The performance is nevertheless adequate for our use case,

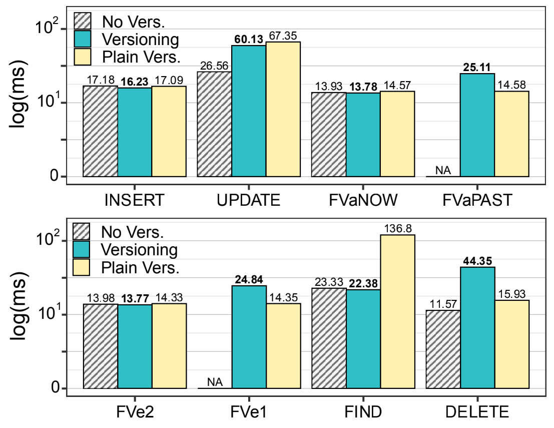

Figure 2: Cloud operation times running the eight operations 100K times in groups of 10K for three different approaches: NO VERSIONING, VERSIONING, and PLAIN VERSIONING.

and even the slowest operation, the UPDATE, needs about 60ms. The three approaches (VERSIONING, PLAIN VERSIONING and NO VERSIONING) show a similar pattern to that seen when using a local DB (Figure 2). The UPDATE is faster than an update using plain versioning with just one main collection. The overhead of versioning in comparison to no versioning during an update is around 33.4 ms. Plain versioning using a single collection gives a faster DELETE than our versioning, by about 28 ms.

As in the local experiment, other operations perform similarly to plain versioning, with the exception of FIND which searches for an unindexed field and causes a collection scan. FIND in our versioning needs as long as a FIND with no versioning, i.e. versioning produces no overhead. However, in plain versioning a FIND is extremely slow, 594 ms, as all versions are kept together and the collection is very large. This confirms our expectation that our solution using a cloud DB has similar performance in most operations and performs better for current documents.

The cloud queries on valid documents show similar execution times with and without versioning. Regarding comparison with plain versioning, Figure 2 shows that our solution takes slightly longer to retrieve past versions (FVE1 and FVAPAST, but performs better when querying currently valid documents which are the most commonly performed operations, resulting in a better versioning performance on the expected use scenario.

The performance of all cloud operations mirrors the local experiment, see [8], so PLAIN VERSIONING performance over time of the FIND operation looks like a staircase, with each step corresponding to a new group of 10K operations (not shown due to space constraints) whereas in our versioning solution the FIND operation performance stays stable (not shown).

Overall, the measurements confirm that the cloud introduces an acceptable overhead and our versioning has a better and more stable performance than using a single collection to store both current and past document versions.

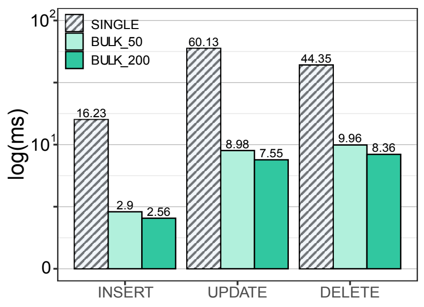

Figure 3: Average cloud execution times of single document (*SINGLE*) and bulk using chunk sizes of 50 (*BULK_50*) and 200 documents (*BULK_200*), performing INSERT, UPDATE, and DELETE, on a log scale.

5.2. Cloud Bulk Operations

Figure 3 compares the cloud use of our versioning solution with single operations (as in previous section) to the use of the new bulk writes using chunks of 50 and 200 operations. The total number of equivalent individual operations is the same, 100K for each operation type executed in groups of 10K.

As shown in Figure 3, the bulk writes (insert, update and delete) are significantly faster than single writes. The improvement is more pronounced for the 200 documents chunk size than when using groups of only 50 documents. The read methods do not have a bulk mode and are not shown. Bulk INSERT is more than 5 times faster than a single insert while the bulk UPDATE gets close to being 8 times faster for 200 documents as compared to single. Bulk DELETE takes four times less time than the single document, even for the 50 document size. Bulk operation times for 200 documents are 8 percent shorter on average than the ones on 50 documents.

6. Conclusions

We presented a new solution for data versioning, tested it locally [8] and in the cloud on MongoDB Atlas and extended it with bulk operations. Versioning performance was satisfactory for our business scenario in both the local and the cloud settings. This good performance can support a flexible and adaptable ERP system and is being used in the system prototype.

We designed and implemented a NodeJS library to manage many document versions by splitting the documents into two collections: live data (main collection) and archival data (shadow collection). Data consistency is maintained by the use of transactions when required, that is for update and delete. Inserts and queries require no transactions. Splitting the data into two collections guarantees good performance on live data, and our tests show clearly that this is superior to keeping old data together with the current data.

We compared the performance of our solution to two alternatives, plain versioning (all data in one collection) and no versioning. We showed clearly that our versioning performs well enough to be used in production, using a local or a cloud DB. Despite a small performance overhead of keeping historical data, we show performance gains on querying current document versions even in non indexed fields, as compared to a solution that does not separate the data into two collections.

The bulk writes we added reduce the number of calls to the DB when inserting, updating and deleting. The performance of bulk operations using groups of 200 documents reduces the network latency overhead significantly, so that the average performance in the cloud is close to the local DB execution using single operations.

Further work is required to investigate the impact of versioning using real business data and processes. Our ongoing work focuses on automated indexing and machine learning for index selection.

Acknowledgments

Thanks to Florian Lang, Christian Dolfus and Martin Sterchi for their contributions. We acknowledge funding from www.innosuisse.ch, Grant No. 44824.1 IP-ICT. We have no conflicts of interest to disclose.

References

[1] P. A. Bernstein, N. Goodman, Multiversion Concurrency Control—Theory and Algorithms, ACM Trans. Database Syst. 8 (1983) 465–483.

[2] M. Stonebraker, L. A. Rowe, The Design of POSTGRES, in: SIGMOD'86, ACM, 1986, p. 340–355.

[3] C. Cioranu, M. Cioca, C. Novac, Database Versioning 2.0, a Transparent SQL Approach Used in Quantitative Management and Decision Making, Procedia Computer Science 55 (2015) 523–528.

[4] H.-G. Kang, C.-W. Chung, Exploiting Versions for On-Line Data Warehouse Maintenance in MOLAP Servers, in: VLDB '02, 2002, p. 742–753.

[5] B. Bebel, J. Eder, C. Koncilia, T. Morzy, R. Wrembel, Creation and management of versions in multiversion data warehouse, in: SAC '04, 2004, p. 717–723.

[6] MongoDB, www.mongodb.com, 2021.

[7] P. Felber, M. Pasin, E. Rivière, V. Schiavoni, P. Sutra, F. Coelho, M. Matos, R. Oliveira, R. Vilaça, On the support of versioning in distributed key-value stores, in: IEEE SRDS, 2014, pp. 95–104.

[8] L. de Espona Pernas, E. Pustulka, Document Versioning for MongoDB, in: ADBIS'22, MegaData, Springer, 2022, pp. 512–524.

[9] E. Pustulka, S. von Arx, L. de Espona, Building a NoSQL ERP, in: ICICT'22, Springer, 2023, pp. 671–680.

[10] E. Sciore, Versioning and configuration management in an object-oriented data model, The VLDB Journal 3 (1994) 77–106.

[11] J. F. Roddick, Schema Versioning, in: Encyclopedia of Database Systems, 2nd Ed., Springer, 2018.

[12] F. Chirigati, J. Siméon, M. Hirzel, J. Freire, Virtual lightweight snapshots for consistent analytics in nosql stores, in: ICDE'2016, 2016, pp. 1310–1321.

[13] U. Störl, M. Klettke, S. Scherzinger, NoSQL Schema Evolution and Data Migration: State-of-the-Art and Opportunities, in: EDBT'20, 2020, pp. 655–658.

[14] D. Sevilla Ruiz, S. F. Morales, J. García Molina, Inferring Versioned Schemas from NoSQL Databases and Its Applications, in: Conceptual Modeling, Springer, 2015, pp. 467–480.

[15] T. Planz, Vermongo: Simple Document Versioning with MongoDB, 2012. URL: https://github.com/thiloplanz/v7files/wiki/Vermongo.

[16] M. Sutunc, Vermongo Mongoose Plugin, 2016. URL: https://www.npmjs.com/package/mongoose-vermongo.

[17] D. Coupal, K. W. Alger, Building with Patterns: The Document Versioning Pattern, 2019. https://www.mongodb.com/blog/post/building-with-patterns-the-document-versioning- pattern.

[18] X. Jin, D. Agun, T. Yang, Q. Wu, Y. Shen, S. Zhao, Hybrid Indexing for Versioned Document Search with Cluster-Based Retrieval, in: CIKM'16, ACM, 2016, p. 377–386.

[19] LearnBoost, Mongoose, 2010. URL: https://www.npmjs.com/package/mongoose.

[20] L. De Espona, Versioning MongoDB Repository, 2021. URL: https://github.com/pier4all/mongoose-versioned.

[21] L. De Espona, Versioning Module MongoDB, 2021. URL: https://www.npmjs.com/package/mongoose-versioned.

[22] L. De Espona, Data Versioning Experiment Repository, 2021. URL: https://github.com/pier4all/data-versioning.

[23] J. Nielsen, Usability engineering, Morgan Kaufmann, 1994.