Adapting TPC-C Benchmark to Measure Performance of Multi-Document Transactions in MongoDB

Asya Kamsky
MongoDB Inc
New York, NY
asya@mongodb.com

ABSTRACT

MongoDB is a popular distributed database that supports replication, horizontal partitioning (sharding), a flexible document schema and ACID guarantees on the document level. While it is generally grouped with "NoSQL" databases, MongoDB provides many features similar to those of traditional RDBMS such as secondary indexes, an ad hoc query language, support for complex aggregations, and new as of version 4.0 multi-statement, multi-document ACID transactions.

We looked for a well understood OLTP workload benchmark to use in our own system performance test suite to establish a baseline of transaction performance to enable flagging performance regressions, as well as improvements as we continue to add new functionality. While there exist many published and widely used benchmarks for RDBMS OLTP workloads, there are none specifically for document databases.

This paper describes the process of adapting an existing traditional RDBMS benchmark to MongoDB query language and transaction semantics to allow measuring transaction performance. We chose to adapt the TPC-C benchmark even though it assumes a relational database schema and SQL, hence extensive changes had to be made to stay consistent with MongoDB best practices. Our goal did not include creating official TPC-C certifiable results, however, every attempt was made to stay consistent with the spirit of the original benchmark specification as well as to be compliant to all specification requirements where possible.

We discovered that following best practices for document schema design achieves better performance than using required normalized schema. All the source code used and validation scripts are published in `github` to allow the reader to recreate and verify our results.

PVLDB Reference Format:
Asya Kamsky. Adapting TPC-C Benchmark to Measure Performance of Multi-Document Transactions in MongoDB. *PVLDB*, 12(12): 2254-2262, 2019.
DOI: https://doi.org/10.14778/3352063.3352140

Proceedings of the VLDB Endowment, Vol. 12, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3352063.3352140

1. INTRODUCTION

MongoDB is a popular non-relational database providing many of the same features as relational databases including ACID transactions[14] and complex ad hoc query language and aggregation syntax[2].

At MongoDB we have not dedicated much effort to standard benchmarks because they frequently measure an extremely limited set of functionality of a database, only directly relevant to a specific workload, and results can easily be misused to decide on suitability of the database for a different workload. In "NoSQL" world the best example of this is YCSB [5] which was created to measure performance of key-value stores and which has extremely limited representation of "typical" data and mainly runs simple read and write operations by primary key, which is not very typical for MongoDB users. In RDBMS world there have been standard established benchmarks overseen by independent organizations[33] for decades, but they haven't been a good fit for "NoSQL" database like MongoDB either, due to the specification requiring normalized relational schema and SQL operations.

However, when we added multi-document transactions to MongoDB in version 4.0 in 2018, we wanted to use a set of performance tests to establish a performance baseline for transactions and to be able to monitor such performance in our continuous integration and test suite [15]. There is a shortage of well defined general workloads to benchmark "NoSQL" databases [27] so we looked to accepted RDBMS benchmarks. A natural candidate for that was one of the better known TPC workloads: TPC-C [9]. TPC-C[32] is meant to emulate a commerce system with five types of transactions involving customers, orders, warehouses, districts, stock, and items represented with data in nine normalized tables. At first glance it may not seem like a good candidate since its requirements are articulated in terms of specific relational schema and SQL statements, but with reasonable number of changes, we believe we created a performance test suite that incorporates MongoDB best practices while executing a workload familiar to those who've worked extensively with TPC-C benchmark and staying consistent with its ACID requirements.

This paper describes in detail the process of adapting a specific existing implementation of TPC-C workload to MongoDB best practices and enumerates things that we discovered which had a positive impact on performance as well as some that failed to produce positive effect counter to our expectations.

Table 1: SQL and MQL operations

SQL	MQL	Comments
SELECT	find	for single collection selects
SELECT	aggregate	for aggregations
INSERT	insert	single or batch
UPDATE	update	single document by default
DELETE	delete	

2. RELATED WORK

Database benchmarking has been an old but enduring topic in academic publications as well as in industry[8]. Many researchers focused on MapReduce paradigms[6] as a natural platform for any parallel processing of "big data", while others applied existing benchmarks and methodology to their own parallel distributed systems[25]. There have been specific proposals to create benchmark specifications in a way that can apply to relational as well as "big data" systems [3] [26]. While some have assumed that YCSB[4] is a defacto standard for "NoSQL" systems, it's limited to testing single isolated read and write operations, all of which rely exclusively on primary key access to documents. There have been proposals to improve YCSB[24] but we didn't find any that included multi-document transactions as an option.

It's common to find assumptions that "NoSQL" access operations are a simple subset of SQL features [28] which may hold for key-value stores, but break down when rich JSON documents are supported extensively.

We also found TPC-C implementations that had been adapted for parallel distributed systems[10] but still assumed SQL and relational schema. Published efforts to use such adapted benchmarks[11] generally ignored either the transactional requirements of the specification or the durability requirement.[29]

An implementation that did not assume SQL was Python-based framework of the TPC-C benchmark for NoSQL systems[1]. It was originally written by Brown University students for a graduate seminar course on NoSQL Systems. The framework was designed to be modular to allow for new drivers to be written for different data stores.

3. SYSTEM MODEL

MongoDB 4.0 replica sets support the following features which allow implementation of the TPC-C specification.

3.1 MongoDB Query Language

MongoDB query language does not map directly to SQL, but it supports a similar set of CRUD operations[19]. Table 1 shows corresponding operations in SQL and MQL (MongoDB Query Language).

In addition, MongoDB supports bulk write operations[16] on a single collection.

3.2 MongoDB Indexes

MongoDB supports primary as well as secondary indexes and can speed up queries by looking up documents in indexes including returning full result from index alone (covered index queries)[20]. Indexes can be created on regular fields as well as fields embedded inside arrays. MongoDB also supports special indexes such as geo-spatial, text search, partial indexes, though none of them were utilized in this work. The query subsystem supports ability to `explain` a query plan which shows which indexes were used for a query (if any) and the number of index entries and documents accessed to produce results.

3.3 MongoDB writeConcern

MongoDB provides a way for an application to specify for any particular write how durable it needs to be and will only acknowledge such write as being successful when the specified level of durability is achieved. In a replica set, the recommended way to ensure durability is to use `writeConcern "majority"` which ensures that when a write is acknowledged as being successful, it's been replicated to majority of the replica set (counting the primary). `Majority` write concern also guarantees the write has been flushed to disk on each of the acknowledging node before acknowledgement is returned.

3.4 MongoDB readPreference

Application can specify which nodes it would like to read data from[21]; by default MongoDB always reads from the primary and only when application specifies a different read preference can a read be directed to a secondary node. A common read preference for applications requiring low latency may be `"nearest"` which means either primary or secondary node can be read from based on the lowest ping time from the client.

3.5 MongoDB readConcern

3.5.1 Read Concern

Read concern setting allows the application to specify what data it wants to be able to read[17]. It is independent from the `read preference` setting. A common setting to use is `"majority"` which means that only data that's been committed to "majority" of the replica set can be seen. This ensures that any data read by client will not be rolled back during an election such as might happen when a primary node fails and an automatic election selects a new primary out of the remaining nodes.

3.5.2 Causal Consistency

An application that's reading from a primary, secondary or either node (whichever is nearest) can switch from one node to another between any two operations. This can happen when a primary fails or a closer secondary is selected by the driver. Even when reads are only of majority committed data, it's possible during such a switch-over to read data that is from time that is not monotonically increasing - i.e. second read could read data that represents state earlier than the first read. To avoid such "back in time" sequence, MongoDB implements causal consistency guarantees in logical sessions[18].

3.6 MongoDB Transactions

MongoDB transactions provide the expected ACID guarantees which TPC-C requires for correctness.[7] A single snapshot of the data is used for the duration of the transaction. A snapshot is a single point in time view of the data at a distinct cluster time maintained via a cluster-wide logical clock[34]. Once a transaction begins with a snapshot at cluster time, no subsequent writes outside of that transaction's context occurring after that cluster time will be seen within the transaction. However, transactions will be able to view their own subsequent writes that occur after the snapshot's cluster time, providing the 'read your own writes' guarantee. Once a transaction starts, its snapshot view of the data is preserved until it either commits or aborts. When a transaction commits, all data changes made in the transaction are saved and visible outside the transaction. Until a transaction commits, the data changes made in the transaction are not visible outside the transaction. When a transaction aborts, all data changes made in the transaction are discarded without ever becoming visible.

This ensures atomicity, consistency and isolation [22]. Our testing included multiple ways of forcing rollbacks of transactions, both while running the benchmark via forcing a node failure and/or failover to another primary in the replica set, and by manually executing part of a transaction without a corresponding commit, while checking consistency of the data.

Within MongoDB transactions, `readConcern` is always set to `"snapshot"`.

Multi-document transactions in MongoDB are committed with an explicit commit statement which specifies desired `writeConcern` or the level of durability required before success is acknowledged. We used `"majority" writeConcern` which in case of a three node replica set means two of the nodes must commit all operations from the transaction for it to be considered successful. Until the commit, none of the writes within transaction can be seen outside the transaction. Only committed writes are replicated to the secondaries where they can only be read after all of them commit.

Transactions immediately obtain locks on documents being written, or abort if the lock cannot be obtained. This ensures that attempts by two transactions to write to the same document will immediately fail for the second transaction at which point it can choose to retry as is appropriate for the application.

4. METHODOLOGY

4.1 Code

We used PyTPCC[1], a Python-based framework of the TPC-C benchmark for NoSQL systems originally written by Brown University students for a graduate seminar course on NoSQL Systems, now maintained by CMU on `github`. The modular framework already had an initial driver implementation for MongoDB which became our starting point.

4.2 Hardware

4.2.1 Database Hardware

For our database hardware, we deployed a three node replica set in MongoDB Atlas[12] - hosted database as a service, which meant that all the configuration was standard and pre-selected once we selected what "size" cluster to use. We chose to use MongoDB hosted service to make it easier for others to reproduce our results, as well as to benefit from built-in monitoring for observing performance of the cluster and its potential bottlenecks. We went with Google Cloud Platform rather than AWS or Azure due to slightly lower cost for comparable hardware and we picked M80 cluster in `us-east1` region to start with which gave us 120 GB RAM and 750 GB storage with 32 vCPUs per node in replica set, which would allow us to test scaling the number of connections up without overwhelming the server. Standard MongoDB replica set deployment is three nodes, one of which acts as the primary, the other two being secondaries and replicating all writes from the primary. In Atlas when all nodes in a replica set are in the same geographic region they are automatically placed in different availability zones [13]. We repeated all the same tests with M60 and M50 replica sets (60GB RAM 16 vCPUs and 30GBs RAM 8 vCPUs respectively). Comparing different size clusters allowed us to compare throughput with the cost of running each cluster.

4.2.2 Client Hardware

For our client machine we selected GCE `n1-highcpu-16` instance in the same region `us-east1` to reduce latency between client and the database. Our monitoring and observations suggested the client was not the limiting factor in any of our tests.

4.3 Configuration

The database configuration was provided to us by Atlas, and no server parameter changes were made.

On the client side, we ran tests varying the following configuration parameters: number of warehouses (scaling factor) the number of client threads, read preference, read concern, write concern (durability setting), causal consistency, and several settings on how PyTPCC should execute, measuring both overall throughput as well as for each transaction the throughput and latency (minimum, p50, p75, p90, p95, p99 and maximum) to make sure we selected reasonable settings to maximize overall performance.

Tests were run with number of warehouses varied from 4 to 1000, number of clients from 1 up to 1000. The database would be loaded once for particular number of warehouses, and then all the tests would be run continuously, duration of permutation of options ranging from minimum of 5 minutes to maximums of over an hour. The only setting we varied in the client configuration was `readPreference`, which we ran as `primary`, `secondaryPreferred` and `nearest`. We varied multiple settings for PyTPCC workload as we made changes to the implementation to measure the impact the changes made and when appropriate introduced a configuration setting to enable running the test with our choice of setting.

4.4 Collecting Measurements

Data was loaded only once, then execution portion was run for many days continuously, varying combination of configuration parameters. For every period of at least five minute, we recorded Tpm-C rate (number of new orders processed per minute, as per TPC-C specification). We also recorded total number of `NEW_ORDER` operations, total number of aborted transactions, total number of retries in a transaction, and total number of all operations both on the client **and** on the server to confirm they match up, as well as output of multiple verification scripts described in the **Verification of Results** section. A sample of a set of metrics representing a single run is included in Appendix D. We did not include in our analysis the collected Tpm-C metrics during runs the verification scripts were running against, as many verification queries are unindexed and inefficient resulting in significant impact on the throughput of the benchmark.

After verifying correctness of each run the summary of results were loaded into its own Atlas cluster where we used MongoDB Charts[23] to generate visual analysis of results and compare different scenarios.

4.5 PyTPCC Code Modifications

Extensive modifications were made to PyTPCC codebase, both in MongoDB driver implementation and other classes.

4.5.1 Transaction Support

Since PyTPCC was written for "NoSQL" systems in 2011, there was no support for transactions in the original implementation even though they are required by the TPC-C specification. None of the "NoSQL" systems, including MongoDB supported transactions at that time. Initially, we added a wrapper around each TPC-C transaction to do each operation in transaction with retries, but later we removed this wrapper from `Stock Level` check as it has no writes and did not require transaction, only correct isolation guarantees. When MongoDB transactions return failure, we test for `"TransientTransactionError"` label since that's expected in several scenarios, like write conflicts (to avoid causing deadlocks) and we continue to retry in a loop on any such error. We added a counter to keep track of, and report the number of retries at the end of each transaction. The number of retries for each transaction type was included in tracked data in summary reports of each run along with other logging, tracking and performance output details.

4.5.2 Standard MongoDB Options Support

Options for authentication, connecting to replica set, setting write concern, read preference, read concern and all other standard MongoDB options were added to configuration file including connecting to MongoDB URI supporting full set of authentication options, as well as read preference and write concern options.

4.5.3 Schema Design

The original implementation provided a normalized option which mirrored the RDMBS schema exactly, and a denormalized option which embedded all customer related information (orders, order lines, history) into the customer document. This is an anti-pattern as the customer record will grow unbounded and performance will get worse and worse the longer the system is operational. We changed things to be more consistent with recommended MongoDB schema practices by restoring most of the normalized model and only embedding order lines inside order documents for the "denormalized" option. This is a very common document database best practice, since it's extremely likely that when we fetch an order we also want to get all of its order lines, and that order lines within each order won't grow unbounded, in fact the array length won't change once the order is "placed". We noted database sizes at the end of the load and each run as well as throughput and latencies. This is where we noted that denormalized schema was **smaller** than normalized, due to every order line not repeating the same information that's in the parent order record. We mention this because conventional wisdom about "denormalization" is that it takes up more space, not less. As performance summaries show, embedding array of order lines into orders provided one of significant performance improvements, along with optimizing indexes. Denormalizing the schema is an obvious performance win since the number of round trips to the database is reduced - you can think of it as pre-joining order lines into the orders table. We show the performance difference in results section Figure 2. While TPC-C explicitly prohibits such schema alterations in its implementation rules[31], this sort of schema design pattern is one of the core advantages of using a document database like MongoDB, and it wouldn't be in line with our best practices to use a different schema.

4.5.4 Optimal Indexes

Next was extensive examination of indexes and replacing most existing indexes with ones that better served the expected queries. Since MongoDB supports both compound indexes as well as covered index queries, this involved examining the query plans for all operations and noting which fields were involved in the predicates, and which needed to be returned. You can see the indexes we settled on in Appendix B.

4.5.5 Reordering and Replacing Operations

Next we examined the code and the logs from each run to understand where additional reduction in latency could be achieved. A number of simple improvements involved redundant queries, unnecessary operations and failure to request only needed fields from the database. Next we looked at causes of frequent retries during transaction attempts: the more operations were performed inside a transaction before a write conflict would cause a roll-back, the more unnecessary work would be undone. We re-ordered write operations to expose a write conflict as early in the transaction as possible, as well as moving such writes before reads where possible, again to reduce the number of operations that would have to be repeated. One pattern that several transactions had was a sequence:

1. Select record "X" from database
2. Update record "X" in database

MongoDB supports an operation `findAndModify`[14] which is equivalent to updating a record and then returning that exact record in a single database roundtrip[1] - either pre- or post-update document can be returned. A variation of this operation is "find and delete" which returns the document that's deleted in the database in the same operation. Replacing two round trips to the database with one of course made the overall benchmark run faster - again while disqualifying the results from "official" consideration for certification, we nonetheless find it more useful to take advantage of operations that we would recommend our users rely on in their real-life implementations rather than artificially constrain the code as the results then would be less useful to the users. We provide the comparison of performance with and without `findAndModify` modification in the results section Figure 3.

4.5.6 Batching Writes

When creating a new order, the specification assumes we will have to perform as many writes to two tables as there are order lines in order, those tables are `ORDER_LINES` and `STOCK`.

[1]in MongoDB "select" equivalent is "find" so this is equivalent to saying do a SELECT and UPDATE in the same operation

When we use denormalized schema, we are inserting order lines into an array inside order object and therefore only do a single insert once the order is completely constructed. When the PyTPCC is run with normalized schema option, or when we are updating the `STOCK` table with information for each item in the order lines, we added a configuration option `batch_writes` and when it's true rather than performing updates one at a time as we loop over order lines, we accumulate them inside an array which we then send to the server once at the end of order creation as a `BulkWrite`[16].

We note here that we observed the effect of batched writes and findAndModify modification was much more prominent when the `writeConcern` setting was `{w=1}` and less so for `{w="majority"}`. The effect of reducing number of round trips to the database would have a bigger impact if the latency of the client to the cluster was measurably larger than the intra-cluster latency.

4.6 Verification of Correctness of Results

We set out to make sure that the code was as compliant with TPC-C requirements as we could make it. We used the following methods of validating correctness of the results and verifying accuracy our metrics.

4.6.1 Asserts in Code

PyTPCC already had asserts throughout the code if any operation didn't match the expected number of records or didn't return a value consistent with specification. We always ran the benchmark in the mode that would throw an error and stop the program when encountering any such conditions.

4.6.2 Verification of Consistency

We created a script of queries representing twelve consistency tests enumerated in section 3.3.2 in TPC-C specification.[30] This was one of the scripts we ran after load, during runs and after test runs.

4.6.3 Verification of Atomicity

We manually performed the atomicity tests described in section 3.2.2 of TPC-C specification.

4.6.4 Verification of Isolation

We performed nine isolation tests described in section 3.4.2 of TPC-C specification.

4.6.5 Verification of Durability

While running the benchmark, we had frequent occasions to test data durability in the following anticipated and unanticipated scenarios. We liberally exercised Atlas `failover` feature where you can click a test button in the UI and simulate a primary "crashing". After bringing down the current primary the service allows one of the secondaries to be elected as the new primary at which point the original primary rejoins the replica set as a secondary and catches up replaying any operations it missed. Using `majority writeConcern` we fully expected that all of our acknowledged (successfully recorded) transactions would be in the server after the failover and comparing the number of expected records in the database with our client output confirmed it.

In addition, since Atlas applies security patches to the OS and minor bug fix versions to MongoDB without prior notice we occasionally encountered an unexpected fail-over during our runs and were able to verify they had no impact on correct reporting of the benchmark. We considered these durability tests to be equivalent to those described in section 3.5.4 of the specification.

We did benchmark the performance of the system with `{w:1} writeConcern` which means the transaction commit returns as soon as the transaction is committed on the primary without waiting for acknowledgement from any of the secondaries. These metrics are included in the complete set of results on `github`.

4.6.6 Verification of Metrics Accuracy

We employed scripted and manual checks to verify the number of transactions reported by the client corresponded to the observed number of records impacted in the database as well as matched the metrics provided by the server.

4.7 Results

We report overall performance for M60 Atlas replica set with `writeConcern "majority"` for durability along with `readConcern "snapshot"` for most transactions, and committed reads equivalent (`readConcern "majority"`, `causal consistency true`) for `STOCK LEVEL` transaction. Figure 1 shows `tpmC` values for different number of warehouses and client threads running. This demonstrates the throughput increasing as we increased the number of threads with color representing varying number of warehouses. Two runs are compared at each thread-warehouse combination, one with `readPreference "primary"` the other with `readPreference "nearest"` and the difference is observed to be minimal since only Stock Level transaction can run on secondary nodes. This was initially a surprising result but given that all reads inside transactions must happen on the primary, we realized we should not have expected to see any performance boost from being able to offload a small number of queries to the secondary nodes.

Figures 2 and 3 show side-by-side comparisons varying a single setting with all other parameters being the same, figure 2 compares performance using normalized vs denormalized schema, figure 3 compares `findAndModify` against corresponding select followed by update.

In figure 4 we map TPMC by number of warehouses for three different instance types in Atlas: M50, M60 and M80. Each is roughly double the capacity of the prior one in amount of RAM and CPU, and M60 is approximately twice the cost of M50 (98%) but M80 is only 85% more expensive than M60. We can clearly see that the increase in throughput does not proportionally correspond to the increase in costs. As with most cloud hosted hardware the price is hourly, we list the exact costs in Appendix C.

4.8 Future Work

Current version of MongoDB (4.0) only supports multi-document transactions on a replica set. The next production release 4.2 will support sharded transactions. As the next step we plan on extending these tests to measure performance in a sharded cluster where the data is partitioned across multiple replica sets.

Once we are able to apply this benchmark to a sharded cluster we plan to scale the data to 10,000 warehouses or more to see how the system scales on very large data sets.

We plan to use this work to test how much secondary reads improve performance when latency between the client

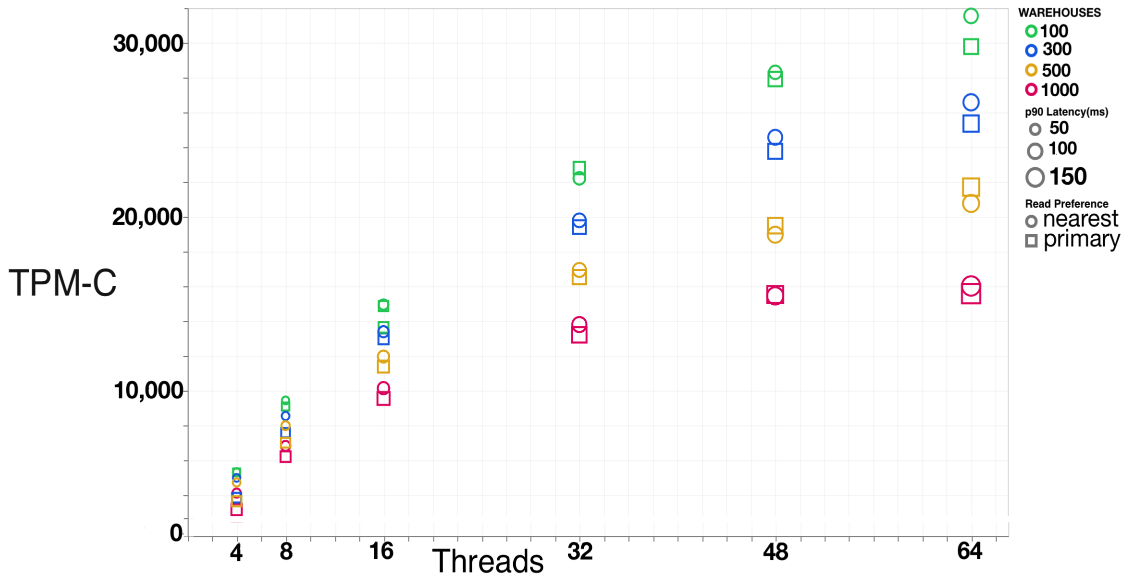

Figure 1: TPMC by number of threads on M60

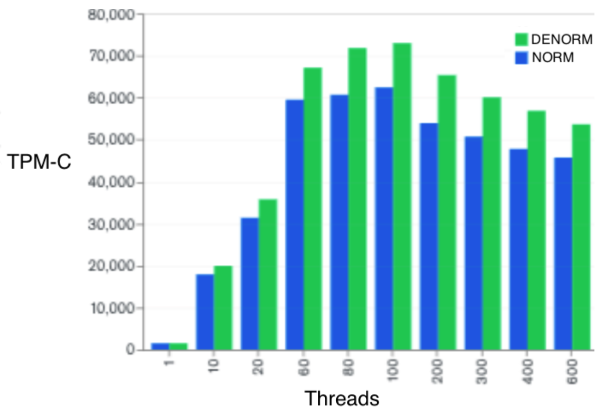

Figure 2: Normalized vs denormalized schema

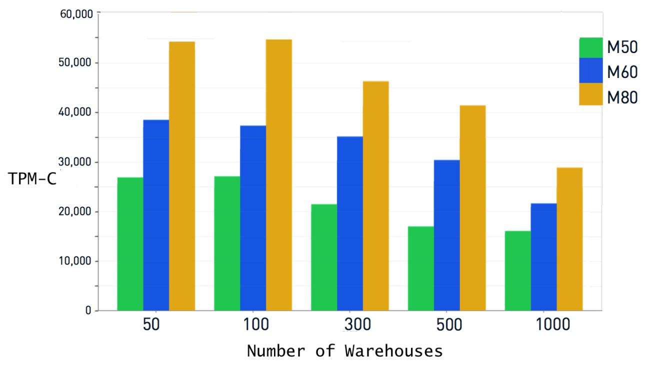

Figure 4: Throughput by Instance Type

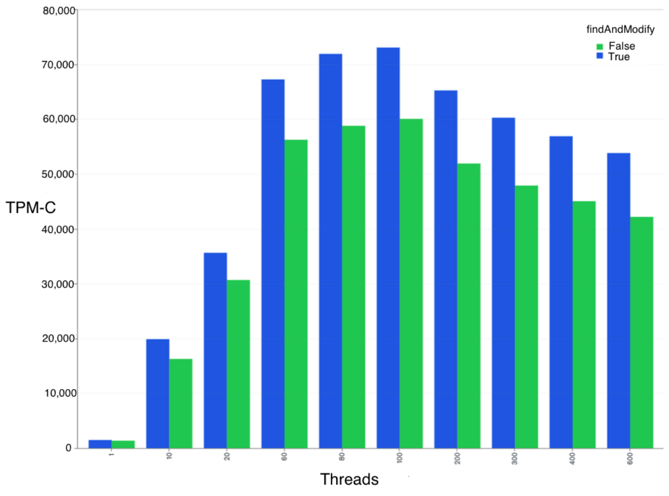

Figure 3: findAndModify vs update+select

and primary node is significantly larger than latency to the nearest secondary - such as would be the case with a globally distributed cluster.

Additional improvements can be incorporated to the verification code, automating some of the process we performed manually, like failing over the primary in the middle of the run based on configured settings and running isolation and atomicity verification scripts automatically during and after each run.

Lastly, we would like to evaluate whether any other TPC benchmarks would provide value if adapted to MongoDB workloads and best practices.

5. CONCLUSIONS

While it is possible to adapt RDBMS targeted benchmarks to document database such as MongoDB with minimal changes to schema and queries, it is likely to produce sub-optimal results unless a significant effort is spent to make extensive modifications to be consistent with best practices of a particular non-relational data platform. Having done that, we believe it is possible to achieve performance that's more representative of document model capabilities and this benchmark will serve us well as we extend functionality of MongoDB going forward.

6. ACKNOWLEDGMENTS

Author would like to thank everyone who contributed to PyTPCC including grad students taking the original Brown University CS227 seminar, Andy Pavlo and CMU for maintaining the repository with the code, Dan Pasette, Andy Schwerin, Cris Stauffer and the performance team at MongoDB for helpful suggestions, Aly Cabral, Grigori Melnik, Michael Cahill, Ian Boros, Stephen Steneker and Pawel Terlecki for proofreading the final result. Any remaining typos are surely introduced by author afterwards.

7. REFERENCES

[1] apavlo. https://github.com/apavlo/py-tpcc/wiki.

[2] E. Botoeva, D. Calvanese, B. Cogrel, and G. Xiao. Expressivity and complexity of mongodb queries. In *21st International Conference on Database Theory (ICDT 2018)*, volume 98 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:23, 2018.

[3] Y. Chen, F. Raab, and R. Katz. From tpc-c to big data benchmarks: A functional workload model. In *Revised Selected Papers of the First Workshop on Specifying Big Data Benchmarks - Volume 8163*, pages 28–43, New York, NY, USA, 2014. Springer-Verlag New York, Inc.

[4] B. Cooper. http://github.com/brianfrankcooper/ycsb.

[5] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[6] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[7] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, June 2005.

[8] J. Gray. *Benchmark Handbook: For Database and Transaction Processing Systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[9] S. T. Leutenegger and D. Dias. A modeling study of the tpc-c benchmark. *SIGMOD Rec.*, 22(2):22–31, June 1993.

[10] D. R. Llanos and B. Palop. Tpcc-uva: An open-source tpc-c implementation for parallel and distributed systems. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, IPDPS'06, pages 330–330, Washington, DC, USA, 2006. IEEE Computer Society.

[11] M. Mitterer, H. Niedermayer, M. von Maltitz, and G. Carle. An experimental performance analysis of the cryptographic database zerodb. In *P2DS@ EuroSys*, pages 6–1, 2018.

[12] MongoDB Atlas. *https://www.mongodb.com/cloud/atlas.*

[13] MongoDB Atlas Manual. *https://docs.atlas.mongodb.com/reference/google-gcp/#regions-with-at-least-three-zones.*

[14] MongoDB Blog. *https://www.mongodb.com/blog/post/mongodb-multi-document-acid-transactions-general-availability*, November 2018.

[15] MongoDB Continuous Integration System. *https://evergreen.mongodb.com/waterfall/mongodb-mongo-master.*

[16] MongoDB Server Manual. *https://docs.mongodb.com/manual/core/bulk-write-operations/index.html.*

[17] MongoDB Server Manual. *https://docs.mongodb.com/manual/core/causal-consistency-read-write-concerns/index.html.*

[18] MongoDB Server Manual. *https://docs.mongodb.com/manual/core/read-isolation-consistency-recency/#sessions.*

[19] MongoDB Server Manual. *https://docs.mongodb.com/manual/crud/.*

[20] MongoDB Server Manual. *https://docs.mongodb.com/manual/indexes/index.html.*

[21] MongoDB Server Manual. *https://docs.mongodb.com/manual/core/read-preference/.* MongoDB Inc, November 2018.

[22] MongoDB Server Manual. *https://docs.mongodb.com/manual/core/transactions/.* MongoDB Inc., June 2018.

[23] MonogoDB Inc. *https://www.mongodb.com/products/charts.*

[24] S. Patil, M. Polte, K. Ren, W. Tantisiriroj, L. Xiao, J. López, G. Gibson, A. Fuchs, and B. Rinaldi. Ycsb++: Benchmarking and performance debugging advanced features in scalable table stores. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 9:1–9:14, New York, NY, USA, 2011. ACM.

[25] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 165–178, New York, NY, USA, 2009. ACM.

[26] P. Pirzadeh, M. Carey, and T. Westmann. A performance study of big data analytics platforms. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 2911–2920, Dec 2017.

[27] V. Reniers, D. Van Landuyt, A. Rafique, and W. Joosen. On the state of nosql benchmarks. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, ICPE '17 Companion, pages 107–112, New York, NY, USA, 2017. ACM.

[28] J. Rith, P. S. Lehmayr, and K. Meyer-Wegener. Speaking in tongues: Sql access to nosql systems. *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, 03 2014.

[29] M. Stonebraker, S. Madden, D. J. Abadi,

S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 1150–1160. VLDB Endowment, 2007.

[30] TPC. *TPC-C Standard Specification, Clause 3: TRANSACTION and SYSTEM PROPERTIES.*

[31] TPC. *TPC-C Standard Specification, Implementation rules 1.4.*

[32] TPC-C. *http://www.tpc.org/tpcc.*

[33] Transaction Processing Performance Council. *http://www.tpc.org/.*

[34] M. Tyulenev, A. Schwerin, A. Kamsky, R. Tan, A. Cabral, and J. Mulrow. Implementation of cluster-wide logical clock and causal consistency in MongoDB. In *Proceedings of the 2019 ACM SIGMOD International Conference on Management of Data*, SIGMOD '19, 2019.

APPENDIX

A. RESULTS OF ALL RUNS

A.1 Best result each config, durable settings

Instance	Warehouses	Threads	tpmC	ops/sec
M50	100	10	11104	414.347
M50	100	20	16260	604.448
M50	100	30	19520	724.677
M50	100	40	21260	791.357
M50	100	50	23027	858.175
M50	100	60	24448	908.332
M50	100	64	25630	952.603
M60	100	4	5336	197.99
M60	100	8	9445	353.693
M60	100	16	14951	556.077
M60	100	32	22802	848.43
M60	100	48	28304	1053.74
M60	100	64	31549	1172.32
M60	100	96	34385	1285.54
M60	300	4	4988	184.93
M60	300	8	8546	318.19
M60	300	16	13391	497.15
M60	300	32	19797	739.027
M60	300	48	24570	914.177
M60	300	64	26585	986.363
M60	300	96	26666	993.347
M60	500	4	4724	174.723
M60	500	8	7989	297.747
M60	500	16	11966	445.353
M60	500	32	16947	628.563
M60	500	48	19491	726.413
M60	500	64	21709	807.123
M60	500	96	22668	845.21
M60	1000	4	4112	152.943
M60	1000	8	6826	256.74
M60	1000	16	10146	379.337
M60	1000	32	13806	513.96
M60	1000	48	15535	577.45
M60	1000	64	16024	598.03
M60	1000	96	16406	611.49
M80	100	64	46449	1733.35
M80	100	84	50924	1892.72
M80	100	94	52329	1945.06

This is sampling of results, due to their large size, the complete set of results are available in the `github` repo at https://github.com/mongodb-labs/py-tpcc.

A.2 Best result for M60, 100 warehouses

Batch	findAndModify	Threads	tpmC	ops/sec
true	true	64	31549	1172.32
true	true	96	34385	1285.54
false	false	64	26959	1001.93
false	false	96	30427	1137.12

B. INDEXES

Tables and Indexes

ITEM
{I_ID:1}
WAREHOUSE
{W_ID:1,W_TAX}
DISTRICT
{D_W_ID:1,D_ID:1,D_NEXT_O_ID:1,D_TAX:1}
CUSTOMER
{C_W_ID:1,C_D_ID:1,C_ID:1}
{C_W_ID:1,C_D_ID:1,C_LAST:1}
STOCK
{S_W_ID:1,S_I_ID:1,S_QUANTITY:1}
NEW_ORDER
{NO_W_ID:1,NO_D_ID:1,NO_O_ID:1}
ORDERS
{O_W_ID:1,O_D_ID:1,O_ID:1,O_C_ID:1}
{O_W_ID:1,O_D_ID:1,O_C_ID:1,O_ID:1,O_CARRIER_ID:1, O_ENTRY_ID:1}

In addition, for normalized schema runs there were additional indexes on ORDER_LINE collection:
{OL_O_ID:1,OL_D_ID:1,OL_W_ID:1,OL_NUMBER:1}
{OL_O_ID:1,OL_D_ID:1,OL_W_ID:1,OL_I_ID:1,OL_AMOUNT:1}

C. ATLAS CLUSTERS USED

Atlas Cluster Costs

Instance	RAM	vCPU	Cost
M50	30GBs	8	$1.66/hour
M60	60GBs	16	$3.29/hour
M80	120GBs	32	$6.13/hour

D. SAMPLE COMPLETE RUN OUTPUT

```
{
  "denorm" : true,
  "batch_writes" : false,
  "write_concern" : "majority",
  "duration" : 300,
  "total" : 327852,
  "tpmc" : 29324.695419676675,
  "causal" : true,
  "warehouses" : 100,
  "find_and_modify" : false,
  "read_concern" : "majority",
  "aborts" : 1480,
  "read_preference" : "nearest",
  "DELIVERY" : {
    "latency" : {
      "p99" : 526.0539054870605,
      "p75" : 193.16411018371582,
      "min" : 96.54998779296875,
      "p90" : 254.01616096496582,
      "max" : 1.7082939147949219,
      "p95" : 296.47088050842285,
      "p50" : 169.342041015625
    },
    "retries_total" : 1048,
    "total" : 13200
  },
  "threads" : 96,
  "date" : ISODate("2019-02-20T23:46:35Z"),
  "ORDER_STATUS" : {
    "latency" : {
      "p99" : 249.05085563659668,
      "p75" : 52.83713340759277,
      "min" : 8.626937866210938,
      "p90" : 95.8399772644043,
      "max" : 0.8874139785766602,
      "p95" : 144.97995376586914,
      "p50" : 31.63313865661621
    },
    "retries_total" : 5692,
    "total" : 12973
  },
  "NEW_ORDER" : {
    "latency" : {
      "p99" : 339.57695960998535,
      "p75" : 99.37191009521484,
      "min" : 29.464006423950195,
      "p90" : 164.55507278442383,
      "max" : 1.9620850086212158,
      "p95" : 198.83394241333008,
      "p50" : 76.35188102722168
    },
    "retries_total" : 5692,
    "total" : 146626
  },
  "retry_writes" : false,
  "STOCK_LEVEL" : {
    "latency" : {
      "p99" : 27.664899826049805,
      "p75" : 12.211084365844727,
      "min" : 6.609916687011719,
      "p90" : 15.162944793701172,
      "max" : 0.16511201858520508,
      "p95" : 18.207073211669922,
      "p50" : 10.541915893554688
    },
    "retries_total" : 25246,
    "total" : 13269
  },
  "PAYMENT" : {
    "latency" : {
      "p99" : 465.4970169067383,
      "p75" : 85.53123474121094,
      "min" : 14.782905578613281,
      "p90" : 172.29509353637695,
      "max" : 2.0620100498199463,
      "p95" : 240.23890495300293,
      "p50" : 43.65897178649902
    },
    "retries_total" : 25246,
    "total" : 141784
  },
  "SERVER": {
    "DELETED" : 132000,
    "INSERTED" : 435075,
    "RETURNED" : 5240273,
    "UPDATED" : 2307556,
    "COMMIT" : 314583,
    "ABORT" : 26726
  }
}
```