Spring 5-14-2020

Benchmarking MongoDB multi-document transactions in a sharded cluster

Tushar Panpaliya

Benchmarking MongoDB multi-document transactions in a sharded cluster

A Project

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Tushar Panpaliya

May 2020

The Designated Project Committee Approves the Project Titled

Benchmarking MongoDB multi-document transactions in a sharded cluster

by

Tushar Panpaliya

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

May 2020

Dr. Suneuy Kim	Department of Computer Science
Dr. Robert Chun	Department of Computer Science
Dr. Thomas Austin	Department of Computer Science

ABSTRACT

Benchmarking MongoDB multi-document transactions in a sharded cluster

by Tushar Panpaliya

Relational databases like Oracle, MySQL, and Microsoft SQL Server offer transaction processing as an integral part of their design. These databases have been a primary choice among developers for business-critical workloads that need the highest form of consistency. On the other hand, the distributed nature of NoSQL databases makes them suitable for scenarios needing scalability, faster data access, and flexible schema design. Recent developments in the NoSQL database community show that NoSQL databases have started to incorporate transactions in their drivers to let users work on business-critical scenarios without compromising the power of distributed NoSQL features [1].

MongoDB is a leading document store that has supported single document atomicity since its first version. Sharding is the key technique to support the horizontal scalability in MongoDB. The latest version MongoDB 4.2 enables multi-document transactions to run on sharded clusters, seeking both scalability and ACID multi-documents. Transaction processing is a novel feature in MongoDB, and benchmarking the performance of MongoDB multi-document transactions in sharded clusters can encourage developers to use ACID transactions for business-critical workloads.

We have adapted *pytpcc* framework to conduct a series of benchmarking experiments aiming at finding the impact of tunable consistency, database size, and design choices on the multi-document transaction in MongoDB sharded clusters. We have used TPC's OLTP workload under a variety of experimental settings to measure business throughput. To the best of our understanding, this is the first attempt towards benchmarking MongoDB multi-document transactions in a sharded cluster.

ACKNOWLEDGMENTS

I want to express my sincerest gratitude to my project advisor Dr. Suneuy Kim for her consistent support and collaboration throughout this research project. I consider myself extremely fortunate to have received an opportunity to work with someone as passionate as her. She has been patient with me since the start of this project and has given me valuable insights that helped me move in the right direction to achieve my goals. I would not have been able to complete this project without her guidance.

I am also thankful to my committee members Dr. Thomas Austin and Dr. Robert Chun, for providing their feedback and guidance. Finally, I want to thank my amazing parents and helpful friends for their countless hours of support and encouragement along the way.

TABLE OF CONTENTS

CHAPTER

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

Scalability is an essential feature of modern database systems. It enables applications to work with massive data without compromising the performance. Sharding in NoSQL databases provides horizontal scalability by data distribution across multiple nodes. It is useful for applications with fast data growth. System design in NoSQL databases is centered around distributed architecture, fast data access, improved performance, and scalability. As stated in the CAP theorem, every distributed system needs to make a trade-off between consistency and availability in a partition tolerant environment [5]. Some NoSQL databases offer higher consistency by compromising availability while others choose eventual consistency.

Applications like System of Records (SOR) and Line of Business (LOB) are types of systems that need support for atomic transactions with ACID guarantees [6]. Relational databases support durable transactions with the highest level of consistency by incurring the cost of data availability and system scalability. NoSQL databases follow a completely different approach and facilitate faster data access and scalable systems to boost overall database performance. Application developers choose NoSQL databases for performance gain but rely on RDBMS for business-critical scenarios. NoSQL systems have started embracing transactions to extend their use cases under critical workloads that demand ACID guarantees.

NoSQL databases like Amazon DynamoDB and Microsoft Azure Cosmos DB offer limited functionality for transaction processing. MongoDB is among the only few NoSQL databases to offer fully compliant multi-document ACID transactions with its all existing distributed system benefits. Multi-document transactions with ACID guarantees are available in MongoDB driver version 4.2 across multiple operations, collections, databases, and shards. Distributed transactions spanning across multiple

documents are useful for running a business-critical workload while working with the horizontally scalable system. The multi-document transaction support enables application developers to work with ACID transactions without losing any distributed system benefits. Benchmarking multi-document transactions in the sharded cluster can empower the system with massive data to run business-critical workloads without compromising scaling benefits.

TPC-A and TPC-C benchmarks are typical OLTP (Online Transaction Processing) benchmarks for relational database systems. These benchmarks run many small transactions to exercise the complexities of a real-world e-commerce system. Since NoSQL systems have started supporting transactions, benchmarking transactional NoSQL systems interest both academia and industry [1]. Especially, there is not much work done for benchmarking the multi-document transaction feature of MongoDB.

MongoDB's research in [7], presents that TPC-C can be used for benchmarking multi-document transactions. It uses a python framework named *pytpcc* that models the TPC-C workload for MongoDB's document-based schema design. *Pytpcc* is a python-based open-source framework of the TPC-C OLTP benchmark for NoSQL systems. It is a simulation of an e-commerce system for running transactions on a backend database and has a MongoDB driver. Previous work [7] on evaluating transaction performance done by the MongoDB community involves running transactions in a 3 node MongoDB 4.0 replication-based cluster along with *pytpcc* framework. However, it does not consider sharding because transactions under the sharded cluster were not available during that time.

Our research expands this idea to a distributed sharded MongoDB cluster. We have extended *pytpcc* framework to work with MongoDB atlas sharded clusters. Our experiments show that multi-document transactions can improve system performance with scaling requirements while maintaining ACID guarantees. The multi-document

transaction being a novel feature to NoSQL systems, we believe our results can address questions about transaction processing amongst developers and NoSQL community.

CHAPTER 2

Related work

CouchDB supports multi-document transactions complying with ACID guarantee in its driver version 6.5. It provides read committed isolation level that guarantees every read performed gets the majority committed data [8]. Transactions in CouchDB are tracked by smart clients to avoid reliance on 2 phase commit protocols. FaunaDB is another NoSQL database that supports transactions across its data partitions. It uses a RAFT-based consensus and Calvin storage engine that enables transaction scheduling and replication [9]. FaundaDB transactions execute in 2 phases – an execution phase and a commit phase. Execute phase reads data from a snapshot, and the commit phase parallelly resolves transaction writes across the data partitions.

Cloud data stores like Google Cloud Storage (GCS) and Windows Azure Storage (WAS) provide the ability to store virtually unlimited data while supporting replication and disaster recovery. These data stores offer high availability in a geographically distributed cluster and provide single item durable consistency [10]. G-store is another key-value data store that supports transactions but only within a group called key groups [11]. Key groups store data, and the keys are cached on the local nodes but can migrate from one group to another to improve efficiency and performance. These data stores implement transaction processing logic within the data store itself. Deuteronomy [12] supports ACID transactions by dividing the storage engine into a transaction component (TC) and a data component (DC). These components function as independent systems that work together to provide atomic operations in cloud-based or local data stores. The ability of components to work independently enables these systems to use heterogeneous data stores.

Systems like Percolator [13] implement the transaction logic inside the client to enable multi-item transactions using a snapshot isolation technique. It relies

on a central timestamp called Timestamp Oracle (TO) and locking protocols to support snapshot isolation. This approach works well for transactions in a single data store but fails to address heterogeneous data stores due to its limiting ability to address deadlocks scenarios [13]. ReTSO – Reliable and efficient design for transaction support is another architecture that uses a centralized system to implement a lock-free snapshot isolation strategy [14]. It reduces the overhead of transaction concurrency and improves throughput measured by concurrent transactions running per minute.

Yahoo Cloud Serving Benchmark (YCSB) is a tool that allows comparing the performance of NoSQL systems [15]. Its key feature is that it can be easily extended by defining new workloads. YCSB is often used to measure the raw performance of the NoSQL system by stressing them under a variety of workloads. OLTP benchmarks, on the other hand, work with many small transactions to measure the transaction performance of the database systems. YCSB+T is an extension of the original YCSB benchmark that allows running operations by wrapping them in transactions. YCSB+T adds 2 novel blocks, a workload executor, and transaction API in the YCSB client [16]. Operations are executed inside transactions in a closed economy workload (CEW). The closed economy workload simulates an OLTP environment but has only a minimal set of operations. The workload executor block executes the workload and validates results by assigning an anomaly score to the workload execution.

YCSB+T uses a client-coordinated transaction protocol [17] across heterogeneous data stores and relies on the datastore's capabilities of single-item strong consistency, adding user-defined data, and global read-only access. These features avoid the need for a central coordinating system to enable transactions to work on multiple items while maintaining ACID properties. The protocol works in 2 phases – In phase one, data items are first fetched from the respective data stores and then tagged with a timestamp in the form of metadata. In phase two, the transaction commits and

updates a global variable called TSR that decides the fate of transactions. Concurrency between transactions is assumed to be handled by the test-and-set ability of individual data stores.

The transaction processing council's TPC-C and TPC-A are a type of OLTP benchmarks that offer multiple transaction types. TPC offers a set of detailed guidelines for systems targeting transaction benchmarking in complex e-commerce environments. These workloads are adopted by traditional relational databases to benchmarks transaction performance to facilitate valuable insights about business-critical scenarios. NoSQL database usage, on the other hand, has been limited to not so critical use cases demanding performance gain. Transaction processing in NoSQL databases is a new area; there is limited research done for benchmarking transactions.

MongoDB's research in [18] suggests that the TPC-C benchmark can be adapted to their document-based NoSQL database. Their research shows promising results but is limited to replica set clusters. Other efforts in [16, 17] suggest that YCSB client can be modified to work with data stores with different data models provided each of these data stored handle ACID guarantees at the individual database level.

CHAPTER 3

MongoDB Sharding and Replication

MongoDB is a distributed document-based NoSQL database that enables application developers to leverage the power of replication, high availability, indexing, and sharding through its schema-less design architecture. MongoDB stores data in JSON documents with support for arrays, nested objects that allow dynamic and flexible schema. It uses powerful queries to enable users to filter and sort information from documents. In this chapter, we describe MongoDB's mechanism of horizontal scaling and synchronization of data across multiple servers.

3.1 MongoDB Sharding

Sharding is MongoDB's mechanism of scaling the cluster capacity to support the deployment of large datasets and improve systems performance. When applications work with huge data, single server capacity can get overwhelmed as the client stresses the server with concurrent operations. There are two different ways of scaling a system: vertical scaling and horizontal scaling. Vertical scaling increases system capacity by adding more hardware components (e.g., RAM, CPU cores, disks) to a single server system. Vertical scaling works well if the dataset has limited growth over time. However, it is inefficient for handling a huge increase in data due to its practical scaling limitations and associated scaling cost.

Horizontal scaling works by adding more commodity computers to the cluster of multiple servers and then distributing the data across these servers. While an individual server cannot handle the entire load, having multiple servers with moderate capacity can work together by distributing the data to improve performance. These servers can be added and removed as per systems need. This approach has higher maintenance costs as compared to vertical scaling but provides more flexibility. Figure

1 depicts MongoDB sharding components

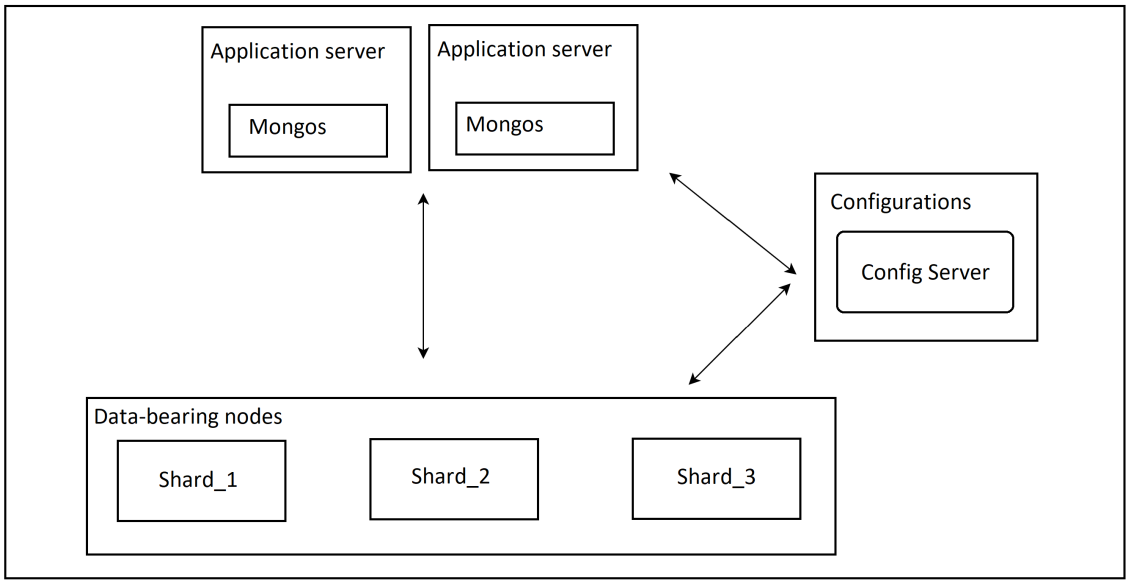


Figure 1: MongoDB sharding components

MongoDB supports horizontal scaling by breaking data into chunks and then distributing them across multiple shards. Each shard can work as a replica set to increase availability. A sharded cluster in MongoDB consists of three components – *shard*, *mongos*, and *config server*. A shard is a data-bearing component that contains a part of system data in the form of chunks. Mongos acts as a middleware between client and server and keeps track of which chunks belong to which shards. Config servers store all the configuration settings about the sharded cluster.

3.2 MongoDB Replication

Replication is MongoDB's mechanism of data synchronization to facilitate data availability, disaster recovery, and data backups. MongoDB replica set consists of 2 or more nodes, where one of these nodes is primary, and others are secondaries. In a replica set, write operations are first applied to the primary node and then recorded in

a capped collection named *oplog*. Each replica set member has its *oplog* that reflects the current database state. Secondary nodes can asynchronously copy operations from the primary's *oplog* to synchronize themselves with the primary node.

The client can vary the durability of write operations using write concern. While data writes are always applied to the primary node, data reads can happen from any member of the replica set. MongoDB allows the client to read data using different read concern values. Read and write concern has a direct implication on data consistency and is explained in detail in the next section. In one of our experiments, we discuss the impact of data consistency on system throughput.

3.3 MongoDB Tunable Consistency

MongoDB provides tunable consistency to the client via different read and write concern values [19]. Read concern value specifies how durable data the application wants to read and write concern determines the level of consistency by specifying how many replicas should acknowledge the write operation to be successful.

3.3.1 MongoDB read concern

Read concern allows the client to set a level of consistency for read operations. Internally MongoDB consults wired tiger storage engine to read data based on the read concern value specified by the client. The default value for read concern in every MongoDB operation is local. While each MongoDB operation can provide its read concern, in case of multi-document transactions, only a transaction-level read concern is used for all the reads that are part of the transaction [20]. It allows transactions to work on consistent data. Table 1 shows available read concern values for MongoDB operations and transactions. It also describes the level of consistency each option

provides.

SR. No	read concern	Description	Rollback possibility	Available to transactions
1	local	Reads most recent data from the node but provides no guarantee that data is majority committed	Yes	Yes
2	available	Reads data from the node if available without guaranteeing it is majority committed	Yes	No
3	majority	Reads data that is majority committed across the cluster	No	Yes
4	linearizable	Reads data from the cluster guaranteeing that all the earlier writes are majority committed	No	No
5	snapshot	Reads data from a consistent snapshot across the cluster	No	Yes

Table 1: MongoDB read concern [2]

3.3.2 MongoDB write concern

When the client issues a write request to the server, MongoDB acknowledges a successful write to the client after it receives the acknowledgments from the number of replicas specified by the write consistency level. Write concern value decides the degree of consistency for the written data. MongoDB allows either an integer value between 1 to N (N = number of nodes) or majority for write concern. Table 2 shows different write concern values available to MongoDB operations and transactions.

SR. No	write concern	Description	Rollback possibility	Available to transactions
1	1	Writes data to only one node from cluster before returning acknowledgement back to the client	Yes	Yes
2	majority	Writes data to more than half data-bearing nodes before returning acknowledgement back to the client	No	Yes
3	n	Writes data to all the data-bearing node in the cluster before sending acknowledgment back the client	No	Yes

Table 2: MongoDB write concern [3]

CHAPTER 4

MongoDB multi-document transactions

MongoDB version 4.0 introduced multi-document transactions with Atomicity, Consistency, Isolation, and Durability guarantees. ACID transactions are useful to satisfy application developer needs for complex scenarios that need all-or-nothing execution while working on consistent data. MongoDB multi-document transactions follow pretty much the same syntax as that of traditional relational database transactions. Figure 2 shows a comparison between MySQL and MongoDB transactions.

MySql transaction

```
START TRANSACTION;

SET @order_id := (SELECT RAND()*10+5);

INSERT INTO new_order(order_id,
                      order_date,
                      status,
                      customer_id)
VALUES(@order_id,
       '2020-01-01',
       'Delivered',
       1002);

COMMIT;
```

MongoDB transaction

```
session.start_transaction()

order_id = Math.rand()*10+5

order = {order_id : order_id,
         order_date = "2020-01-01",
         status = "Delivered",
         customer_id = 1002}

new_order.insert_one(order, session = session)

s.commit_transaction()
```

Figure 2: MySQL vs. MongoDB transaction

The introduction of multi-document transactions amalgamates MongoDB's document-based model and distributed ACID guarantees. We now address the design details of MongoDB multi-document transactions. These are essential features that empower multi-document transactions to work with highly consistent data while maintaining system integrity. MongoDB multi-document transactions are empowered by the following features [6]. The rest of the chapter summarizes these features that are considered to be crucial to understanding MongoDB multi-document transactions.

- Efficient resource allocation through logical sessions
- Improved read isolation using local snapshots
- Enhanced synchronization of sharded replica using the hybrid clock
- Update-structure of Wired tiger storage engine
- Durability supported by retryable writes
- Efficient reading mechanism using safe secondary reads

4.1 Efficient resource allocation through logical sessions

In MongoDB (version 3.6+), every operation can be linked to a client session, namely causally consistent sessions. A causally consistent session is represented by a unique identifier denoted by *lsid*, which consists of a GUID (Globally Unique ID) and a user id. It is associated with a client during its communication with a MongoDB cluster. Every resource that is used by the client is attached to a unique session [21].

In multi-document MongoDB transactions, a single transaction can use multiple resources before it runs to completion. A centralized system to manage these resources can become a bottleneck of single-point failure and thus can be very inefficient. The causally consistent sessions, on the other hand, provide much more flexibility. These sessions facilitate resource-intensive processes involved with multi-document transactions such as resource tagging, garbage collection, and resource cancellation.

4.2 Improved read isolation using local snapshots

MongoDB runs queries on the server by acquiring a lock on the documents. Each query yields after a periodic interval to make sure a single operation does not hold a resource for too long. A query on the MongoDB server is processed by fetching data before the query starts and then periodically saving data and locks related to that

query at the time of yielding. The server erases the old state of data for the query and acquires a new state when the query resumes its execution the next time. Local snapshots provide server an ability to retain all the data and locks when the query yields [22].

Local snapshots are essential in the context of multi-document transactions because they provide an ability to keep track of all the resources used by a transaction. They offer transaction an ability to work on data from a given point in time. It is a vital feature to guarantee the atomicity of the transaction. When a transaction starts within a logical session, the client needs to use read concern "snapshot" to use consistent data until the transaction commits or aborts [6].

4.3 Enhanced synchronization of sharded replica using the hybrid clock

In MongoDB distributed cluster, local clocks for each shard tracks the *oplog* entries to provide order. Ordering is achieved by recording the logical timestamp of each durable write operation. Secondaries fetch these ordered entries and replay them to synchronize themselves with primary nodes state. Multiple shards with individual local clocks suffer a considerable cost of synchronization in providing consistent data to multi-document transactions.

A hybrid clock approach helps to solve this problem by combining the operations count and system time to create a special type of timestamp called a hybrid timestamp. This timestamp is then exchanged in the cluster through the gossip protocol. When a node receives a message containing this timestamp, it updates its local timestamp with the received timestamp if it is later than its timestamp [23]. The hybrid timestamp is hashed using the private key of primary nodes for tamper prevention. The hybrid clock helps multi-document transactions to work on

synchronized data spanning across multiple shards. It is crucial because, in most scenarios, multi-document transactions access data across multiple shards.

4.4 Update-structure of Wired tiger storage engine

Wired tiger is the default storage engine of MongoDB that stores data in a key-value tree structure. It comes with a mechanism to maintain point-in-time data for the transaction snapshots. The storage engine maintains an update-structure and adds a key-value entry to this structure every time a permanent change is made. The key-value entry holds the timestamp of the update, state of data, and a pointer to the next update-structure for that same data. These update-structures help MongoDB transactions to query data at a specific point in time and re-synchronize in case of rollbacks [24].

The storage engine's ability to provide point-in-time data and handle multi-version concurrency control enables multi-document transactions to reduce data locks. Wired-tiger update structure also facilitates snapshot retention and data rollbacks in failure cases.

4.5 Durability supported by retryable writes

In MongoDB, when a client performs a write operation, the server has the responsibility of writing the update on to the database and sending an acknowledgment back to the client. Acknowledgment is sent back based on the write concern value provided by the client. If any of the replicas involved with this write operation does not acknowledge this write due to node failure or a network failure, MongoDB cannot acknowledge the client. In this situation, the client may resubmit the same write request considering the previous write failed. If the previous write operation happens

to make to the replica, the first and the second writes conflict, and a dirty write situation can arise.

Retryable write gives the server an ability to retry a failed write automatically. The primary node in the cluster maintains a special type of table called the transaction table. The table contains a list of logical session ids, transaction id, and a pointer to an *oplog* entry where the operation is recorded. When a write request comes to the primary node, it checks if the transaction id is already present in the table, if found the server identifies it as a retryable write.

In the case of multi-document transactions, commit and abort operation are retryable by default [25]. Transactions maintain their atomicity by retrying the entire transaction in case of any failure.

4.6 Efficient reading mechanism using safe secondary reads

In a sharded cluster, chunk migration between shards happens as a part of load balancing across nodes. The load balancing process first identifies the chunks to migrate and then copies those chunks from source shard to destination shard and then deletes the data from source shard. A special type of routing table at primaries of shard identifies and filters out documents that are in the middle of migration. While this works for queries when routed to primaries of shards, reads from secondaries can read inconsistent data as they are unaware of the migrating documents.

A safe secondary reads mechanism addresses this issue by copying the routing table to secondary nodes [26]. Replication of routing tables on secondary nodes helps to make secondaries aware of chunk migration. The addition of safe secondary reads helps multi-document transactions read data consistently from the cluster using read concern local or majority.

CHAPTER 5

OLTP benchmarks

Online transaction processing (OLTP) is a type of information processing that supports transaction-based data manipulation for a database system. It is usually associated with applications like online-banking, e-commerce, airline reservations, manufacturing, and shipping systems. OLTP benchmarks, such as TPC-C, TATP, SmallBank, and SEATS, are most concerned with the atomicity of the database operations to support concurrent insert, update and delete operation in the system [27]. The main goal of these benchmarks is to provide a real-world client-server system where a large group of users can perform essential transactions in a distributed decentralized database system. Data in OLTP workloads reflects real-life business interactions in an organization.

The database schema for TPC-C is based on an RDBMS table-based structure. We choose the TPC-C benchmark to measure the performance of multi-document transactions as it is an industry-standard OLTP benchmark. It's complex warehouse-centric design, and heavy update transactions make it suitable for transaction benchmarking in MongoDB.

5.1 TPC-C benchmark

TPC-C benchmark consists of multiple transactions that simulate complicated e-commerce system operations. Transactions in the TPC-C benchmark workload are a mix of read and heavy update operations. The purpose of this benchmark is to provide standard guidelines for systems and a set of diverse operations that can simulate an OLTP application irrespective of underlying hardware configurations. It differs from other benchmarks that are limited to specific machines or an operating system. The overall performance of the system is a measure of the number of new orders processed,

which is committed transactions per minute denoted by tpmC. With its pervasive mix of read-write operations and the complexity of the transactions, the throughput is also considered business throughput. The following factors characterize the TPC-C benchmark

- `Concurrent execution of transactions with varying complexity`

  A mechanism to stress the system with multiple clients using its 5 different transactions that exhibit varying read-write load on the database

- `Real-time as well as queued execution mode`

  An ability to defer execution of transactions by queuing them lets the benchmark simulate realistic transaction events

- `A diverse system with a variety of relationships and attributes`

  A table structure designed with a set of complex attributes and the relationship between entities

- `Transaction integrity through ACID`

  A Strongly consistent table-based design that maintains atomicity, consistency, integrity, and durability of transactions

5.2 TPC-C schema design

TPC-C benchmark is composed of a variety of complex operations designed for portraying a real-world e-commerce system activity [4]. These operations need a schema design that can support non-uniform data access while working on data with a variety of relationship and sizes. TPC-C benchmark simulates realistic transaction in an e-commerce system that uses geographically distributed regional warehouses. The number of warehouses determines the total data size. Each warehouse stocks 100,000 items and has the responsibility of facilitating sales activities in 10 sales districts.

Each of these districts is home to 3000 customers. Figure 3 explains TPC-C's system hierarchy.

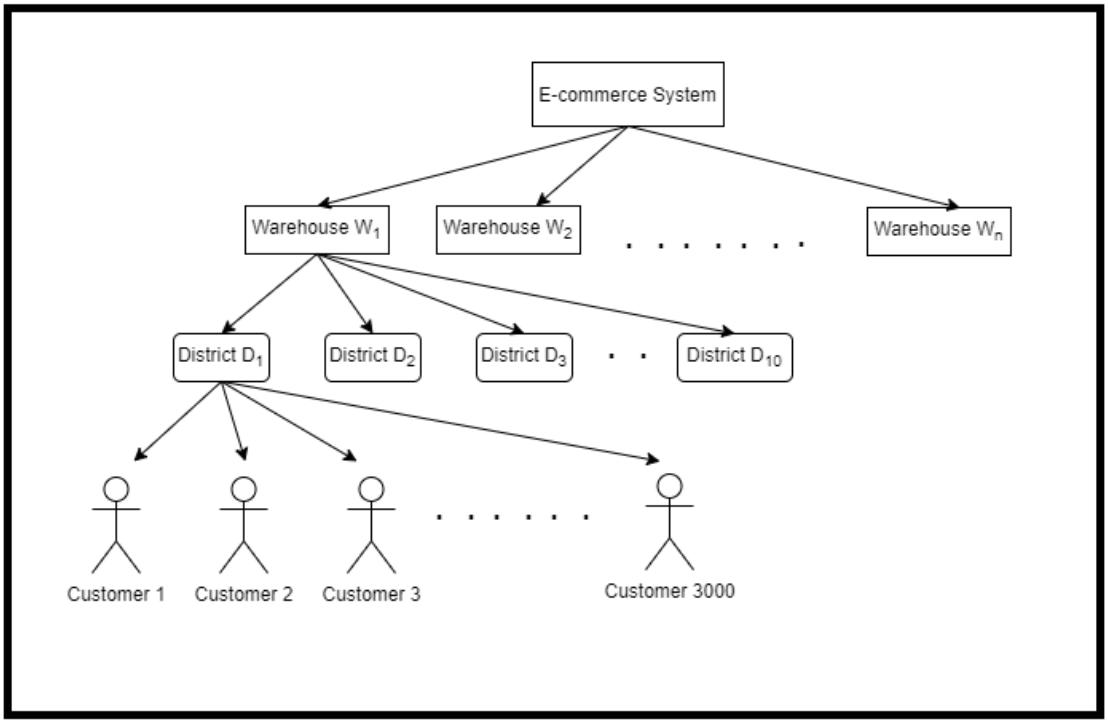


Figure 3: TPC-C database system heirarchy[4]

5.3 TPC-C Database Schema

The TPC-C database consists of the following nine tables:

1. `Warehouse` - The warehouse table stores information about each warehouse on the system.

2. `District` – The district table stores sales information of a district.

3. `Customer` - Customer table stores customer's personal and sales information.

4. `History` - History table stores past order information of customers.

5. `New order` - New order table stores order information about every new order placed.

6. `Order` - Order table stores order information about each order.

7. `Order line` - The order line table contains information about each item in the order.

8. `Item` - Item table represents an instance of stock and stores its information.

9. `Stock` - Stock table stores information about each item at all the districts and warehouses.

ER diagram in figure 4 shows the relationship between the 9 tables in the database system.

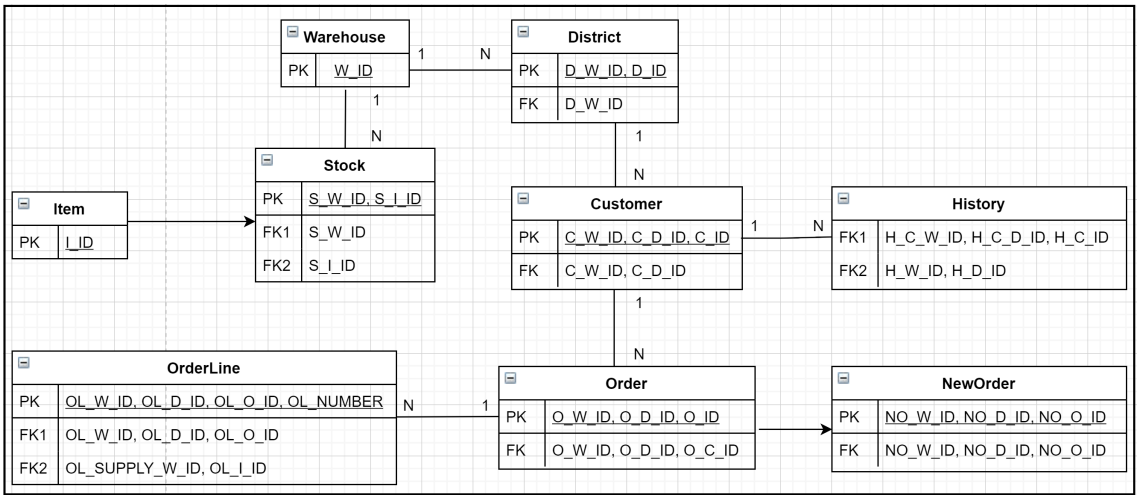


Figure 4: TPC-C database system ER diagram [4]

5.4 TPC-C transactions

TPC-C benchmark runs 5 different transactions of varying complexity, as presented in Table 3. A framework adapting TPC-C should support transaction execution in real-time as well as deferred mode via queuing. The ability to execute the transaction in these modes gives the framework an ability to exercise transactions with varying read-write distribution. The workload is expected to maintain a minimum percentage of each transaction type over the run duration.

SR. No	Transaction type	Minimum percentage
1	Order-status	4 %
2	Delivery	4 %
3	Stock-level	4 %
4	Payment	43 %
5	New-Order	No Minimum

Table 3: Minimum percentage of different transaction type [4]

5.4.1 New-Order

This transaction simulates an order placement scenario as a single atomic database transaction. TPC-C benchmark uses the number of new orders processed per minute for performance measurement. It is designed to put the system under variable load to reflect a real order placing scenario in a production system. The transaction is implemented in a way that one percent of all the new orders contain an order item that produces a data entry error at the chosen warehouse and thus needs to be imported from another warehouse. If an order contains items imported from some warehouse, it is considered remote order. Otherwise, it is considered a local order.

5.4.2 Payment

This transaction updates payment-related data for a randomly chosen customer. It involves updating customer's balance, district payment information, and warehouse payment statistics. It is relatively light weighted as compared to other transactions. In these transactions, customers can be chosen by their last name to perform non-primary key access.

5.4.3 Order-status

Order-status transaction checks the status of the customer's last-placed order. It is a read-only transaction with a shallow frequency. It also retrieves customer information by a non-primary key access strategy.

5.4.4 Delivery

Delivery transaction runs in a deferred mode. It involves processing up to 10 orders in a batch and delivering them. It works via queuing requests and then executing in a batch.

5.4.5 Stock-level

It is a read-only transaction that checks if recently sold items are below the user threshold in the warehouse. It involves a heavy read-load on the system and has a shallow frequency.

CHAPTER 6

Pytpcc framework

Pytpcc is an open source python based TPC-C OLTP framework designed for NoSQL databases. It was first developed by students from brown university and then adapted by MongoDB for benchmarking multi-document transaction in 2019 [28]. The framework is extensively designed to support writing new NoSQL drivers with ease. A new driver can be added by extending frameworks abstract driver (*abstractdriver.py*) and implementing functions to load data and run workload specific to the database system. Pytpcc has drivers for various NoSQL databases, including MongoDB, Cassandra, HBase, CouchDB, and Redis. The framework is divided into below 3 packages –

1. `Driver` -- This package contains driver implementations for different NoSQL systems.
2. `Runtime` -- This package contains classes used for loading the data and executing the workload.
3. `Util` -- This package contains utility classes used by the benchmark.

The framework executes the benchmark in two phases – a loading phase and an execution phase.

6.1 Load Phase

In load phase database is populated based on the configuration options provided to the driver class. If –no-load option is used in the configuration, then the load phase is skipped; otherwise, the driver's *loadStart()* function is executed, which prepares the driver to start loading. Next, a random number of tuples and a table name is sent to

the drivers *loadTuples()* function to populate the table. *loadTuples()* can be called multiple times and runs until all the tuples are loaded. Once all the data is populated *loadFinish()* function notifies the controller that loading is finished.

6.2 Execution Phase

In the execution phase, the workload runs a randomly chosen transaction on the data. Once a transaction type is chosen, the associated method is called that runs a set of operations as a single transaction. There are two optional functions in this phase that can be used to prepare and commit the transaction. The activity diagram in Figure 5 shows a sample *pytpcc* benchmark execution.

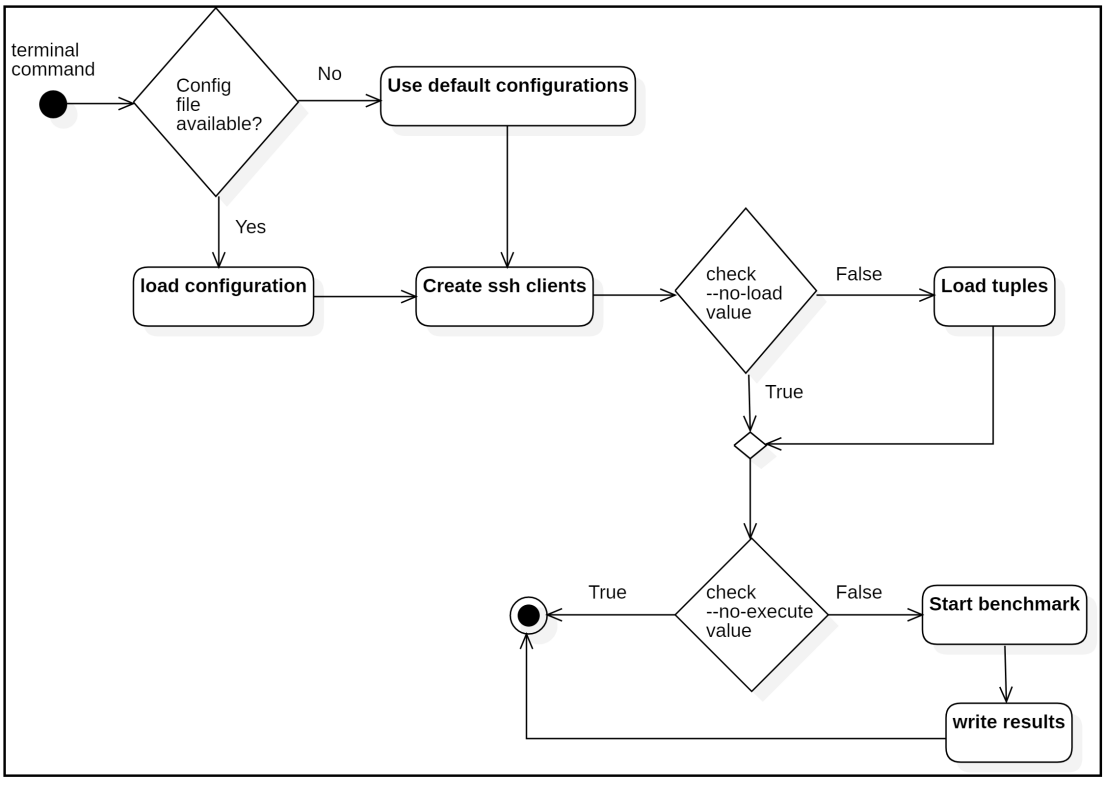


Figure 5: Pytpcc Activity diagram

6.3 MongoDB driver

MongoDB driver for *pytpcc* was improved by a team at MongoDB to benchmark transaction performance [18]. It extends the abstract driver and implements its function for the MongoDB database. The driver implements a load and execute phase for multi-document transactions. It also supports providing configuration parameters to the driver from both a configuration file and command line.

6.4 Configurations

This section describes the configuration options available in the framework. These configurations allow the client to provide a custom setting to the workload. Below is the list of available options and their interpretation.

6.5 Database Schema design for MongoDB

In traditional relational databases, tables are normalized to eliminate various anomalies caused by duplicate data. Therefore, join operations plays a crucial role in gathering relevant data from the normalized table. However, join operations inherently clashes with clustered environments. In distributed NoSQL databases such as MongoDB, it is costly to join normalized data distributed over multiple servers in a cluster. Therefore, data are denormalized for queries to find the required data ideally in one place at the cost of data duplication. In MongoDB, data can be modeled in two different ways: embedding and referencing. Embedding supports nested documents and considered the right choice when documents exhibit a one-to-many relationship. Referencing models data with references between documents and useful for hierarchical data sets.

TPC-C is a benchmark developed for relational database systems, and therefore,

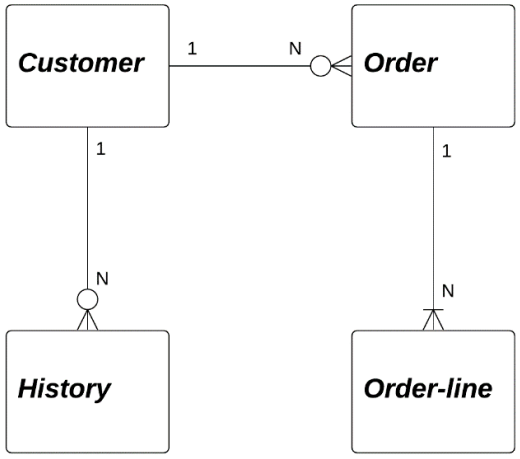
SR. No	Configuration	Value	Interpretation
1	config	string	Used to provide a configuration file for the driver. If no configuration is provided, default configurations from the drivers are used
2	warehouses	integer	Used to set the number of warehouses to be used in the workload
3	duration	seconds	Represents the number of seconds for which the workload should be executed
4	denormalize	boolean	If this option is set to true, the system uses an embedded schema design. By default, this value is set to false that represents referenced schema design
5	no-load	boolean	Used to skips the loading phase. The framework starts executing the workload on already populated data
6	no-execute	boolean	Used to skip the execution phase
7	debug	boolean	Used to print log messages to console
8	scale-factor	float	Used to scale the system size. If a value greater than 1 is used, it will scale down the database by that factor
9	clientprocs	integer	Represents the number of concurrent clients to be used for the workload
10	clients	string	Represents ssh client to be used for running the workload
11	uri	string	Used to specify MongoDB atlas connection string
12	print-config	boolean	Used to print the default configurations of the framework
13	ddl	string	This option enables the user to provide a file for the TPC-C data definition language
14	findAndModify	boolean	Uses findAndModigy atomic queries instead of find and update

Table 4: Pytpcc configurations

tables are normalized in the TPC-C database. Pytpcc framework provides both normalized and denormalized schema for the same data set. The following section presents details about referenced and embedded schema design in pytpcc.

6.5.1 Referenced schema of Pytpcc framework

In the case of referenced schema design, each table is considered as a separate collection. All the insert, update, and delete operations are done on a single document at a time. This design is a direct extension of a normalized relational database schema. Figure 6 depicts the schema and a sample document based on referencing. A Complete document for the referenced schema is available in Appendix A.




```
CUSTOMER: {
   "C_ID": 1,
   ....
   ....
}

ORDER: {
   "O_ID": 2452,
   ....
   ....
}

ORDER_LINE: {
   "OL_O_ID": 3675,
   ....
   ....
}

HISTORY: {
   "H_C_ID": 2,
   ....
   ....
}
```

Figure 6: Pytpcc Referenced Schema

6.5.2 Embedded schema of Pytpcc framework

In embedded schema design, information related to a given customer is embedded in the customer document itself. This related information includes customer orders, order lines for each order, and history. MongoDB suggests embedding as the preferred

denormalization method. Figure 7 depicts the schema and a sample document based on referencing. A Complete document for embdedded schema is available in Appendix B.

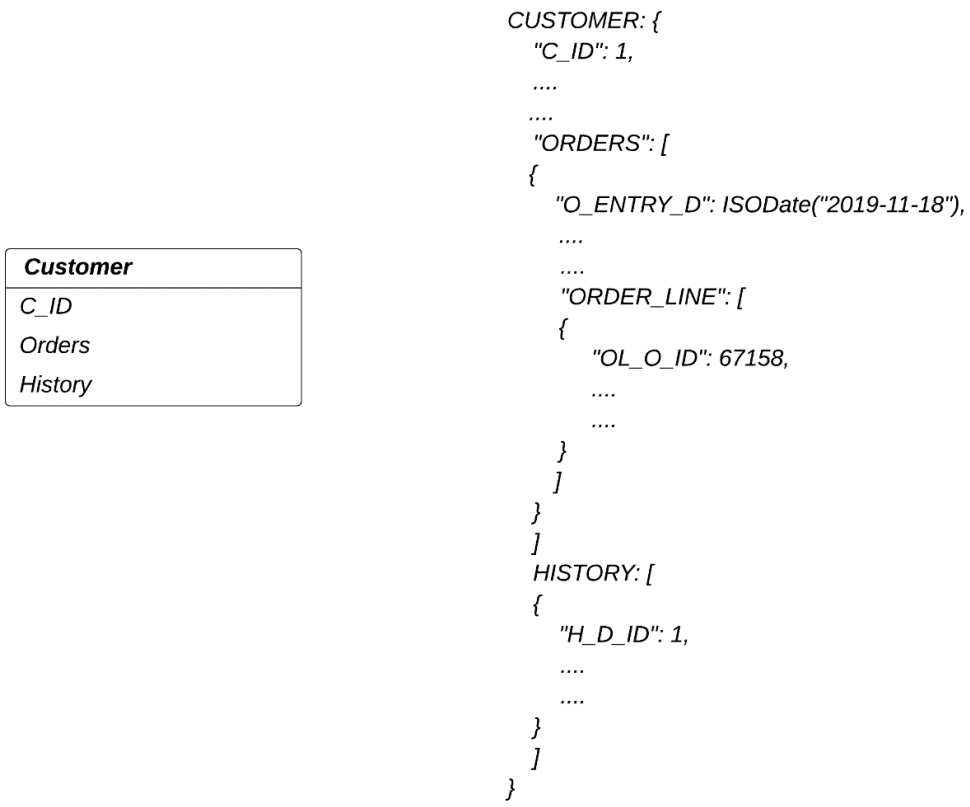


Figure 7: Pytpcc Embedded Schema

CHAPTER 7

Experiment and Results

This chapter presents the benchmarking experiments conducted to evaluate the performance of multi-document transactions in MongoDB sharded clusters. It also covers our analysis of the experimental results. The data gathered from these experiments address the following topics

- *Impact of read and write consistency levels on the performance of multi-document transactions running in a sharded cluster*
- *Impact of sharding on system scalability*
- *Schema design preferences for OLTP applications running on a sharded cluster*
- *Impact of data size on the overall throughput in a sharded cluster*
- *Impact of atomic findAndUpdate operation on systems performance*

7.1 Experiment setup

We performed all the experiments on a *c5n.4xlarge* instance on AWS with *ubuntu 16.4 LTS (HVM)* installed as a client and MongoDB atlas M40 cluster as a server. M40 cluster tier comes with a server setup consisting of 16GB RAM, 80GB of storage, and 4 vCPU. It supports network performance bandwidth of 10 Gigabit with up to 6000 connections. We choose AWS as a background resource provider for the atlas server. MongoDB atlas enabled us to create clusters with ease and modify configuration based on the need. The Client machine setup involved cloning *pytpcc framework* and installing *pymongo*, *dnspython*, *execnet*, and *argparse* libraries.

7.2 Experiment 1 - Impact of read-write consistency on throughput

MongoDB provides the user with an option to tune the consistency level by configuring read and write concern values in the client code. This experiment is to see

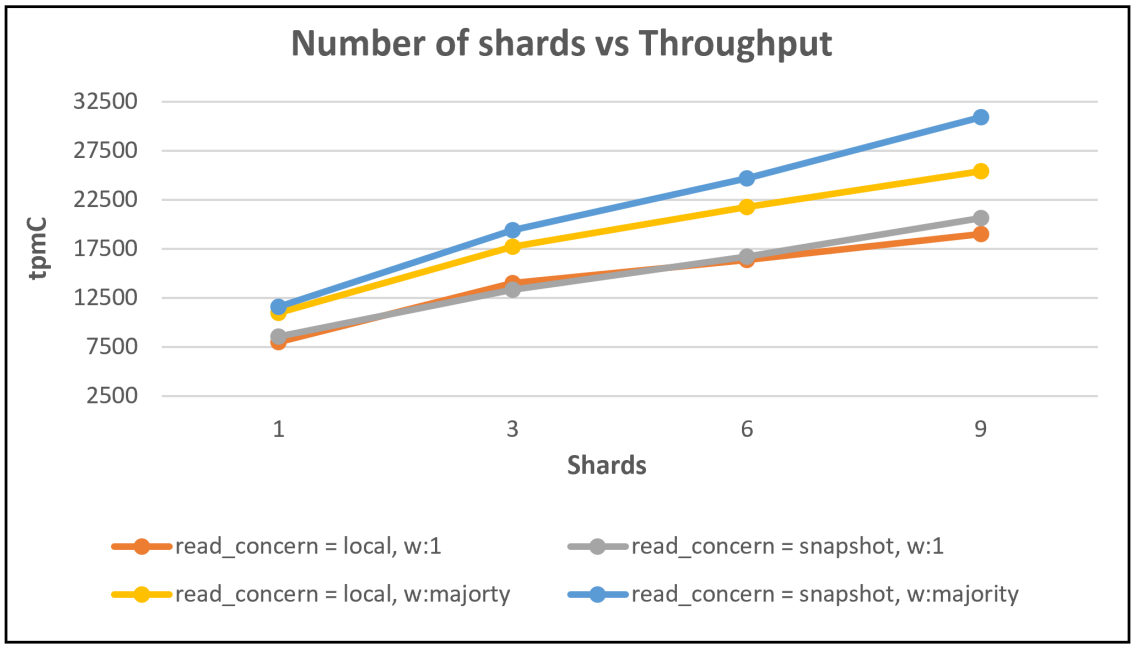


Figure 8: Impact of read-write consistency on throughput

the impact of read and write consistency level on the throughput of multi-document transactions. In MongoDB, transaction read_concern can be set to *local*, *majority*, or *snapshot* where snapshot provides the most consistent data for read operations of multi-document transactions running across the shards. write_concern can be set to any numeric value between 1 to N, where N represents the number of nodes in the cluster. We performed this experiment on systems with 1, 3, 6, and 9 shards for testing scalability. First, each cluster was populated with 100 warehouses data, and then the benchmark was executed for 10 minutes by increasing the number of concurrent clients until throughput saturated. tpmC reading was recorded for each experiment. Figure 8 shows throughput saturation points for various read-write consistency settings over the number of shards in the cluster.

Observation 1 – In figure 8, orange line (read_concern = local, w:1) and gray line (read_concern = snapshot, w:1) show that read concern does not affect the

throughput much when write concern value is set to its lowest (w:1).

Rationale – w:1 provides the lowest durability for write operations and acknowledges to the client as soon as data is written to a single node. During the workload execution, most transactions are aborted at commit time due to non-durable write operations. Snapshot read concern also does not increase throughput due to the write inconsistencies at the commit time.

Observation 2 – An increase in the write concern value increased systems throughput.

Rationale – In MongoDB multi-document transactions, data is written to the nodes at commit time, which saves latency in replicating individual writes of the transaction as and when they happen. This reduction in latency improves systems throughput. When transactions execute with strongly consistent write concern, at commit time, all the writes are guaranteed to persist on the disk. The consistent state of the database boosts overall transaction throughput.

Observation 3 – In Figure 8, yellow-line (read_concern = local, w: majority) and the blue line(read_concern = snapshot, w: majority) shows that, read concern has a significant impact on the throughput when majority write concern is used.

Rationale – In cases when w: majority is used, writes are guaranteed to persist on the majority of the nodes. While read concern = local reads the most recent data available to the server, it does not provide any guarantee that the data that is being read is consistent. On the other hand, read_concern = snapshot provides a guarantee that the read data is consistent across shards. This improves the number of successfully committing transactions and therefore resulting in improved throughput.

Result Analysis - Results obtained from this experiment show us that multi-document transactions in sharded clusters need a reliable write consistency for optimal performance. Snapshot reads across shards are an essential addition to multi-document transactions and can guarantee the highest level of consistency without compromising throughput. Sharding scales the transaction throughput with consistent read and write concern.

7.3 Experiment 2 - Impact of schema design on throughput

This experiment tests impact of the design choice on the system's overall throughput. We performed this experiment in an M40 sharded cluster with 3 shards and populated the database with 100 warehouses in both referenced and embedded schema design. Benchmark was executed for a fixed duration of 10 minutes by increasing the number of concurrent clients.

In referenced schema design, the system is populated with a separate document for each customer, order, order line, and history. Orders that belong to a customer are referenced by customer id in order document. Each order line is represented by a separate document and contains a reference to the order that it belongs to. Figure 6 of Chapter 6 depicts the database schema and sample document.

In embedded schema design, all the orders belonging to a customer are embedded in the customer document. The customer document is also embedded with an array of past orders that belongs to the customer. Figure 7 of Chapter 6 depicts the database schema and sample document.

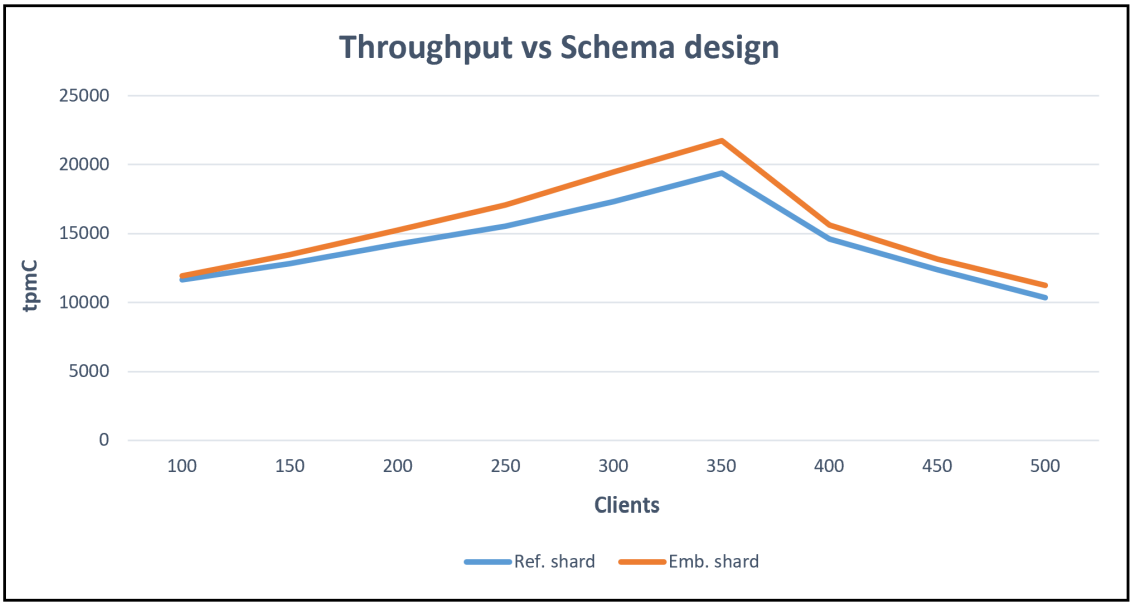


Figure 9: Impact of schema design on throughput (Ref. Shard: Sharded cluster with referenced schema, Emb. Shard: Sharded cluster with embedded schema)

Observation 1 – As seen in Figure 9, embedded schema improved throughput by approximately 12% as compared to the referenced schema.

Rationale – For a new order transaction, an average of 10 order lines are chosen for each order. Embedded schema design accesses fewer documents as compared to referenced schema design and therefore increasing overall throughout.

Observation 2 – The throughput of the system saturated at 350 concurrent clients.

Rationale – In experiment 1, when we worked with 100 warehouse data in the M40 cluster, the system throughput also saturated at 350 concurrent clients, and we observed that this saturation point increases with adding more shards in the system.

Result Analysis – Results obtained from this experiment show that multi-document transactions, when used with document embedding design approach, can guarantee ACID properties for the transaction with improved performance.

7.4 Experiment 3 - Impact of sharding on throughput with various data size

This experiment is focused on finding the impact of sharding on the throughput of multi-document transactions. We varied the data size of the system in a 3 shard M40 cluster by changing the number of warehouses in the system. We start with a system having 1 warehouse and increase the number of warehouses till 500 under the same cluster settings. We stress the system by the increasing number of concurrent clients to reach throughput saturation. The graph in Figure 10 shows the throughput measured from an unsharded and sharded replica set while varying data size

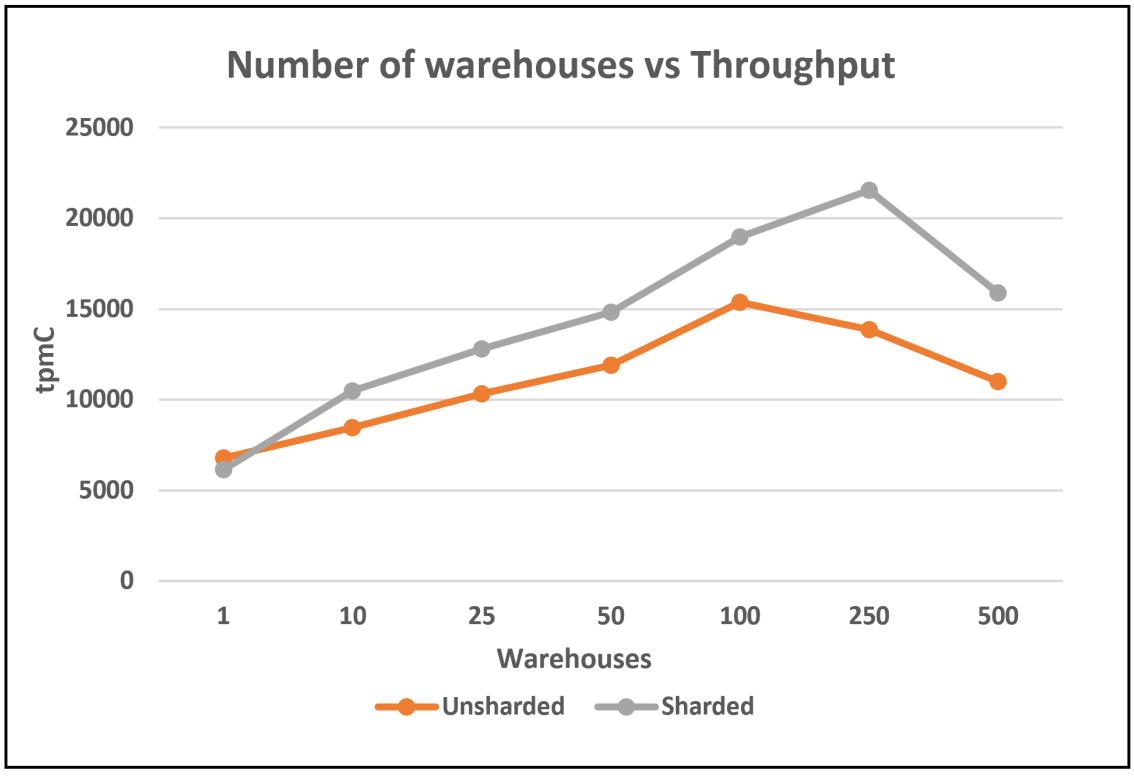


Figure 10: Number of warehouses vs. Throughput

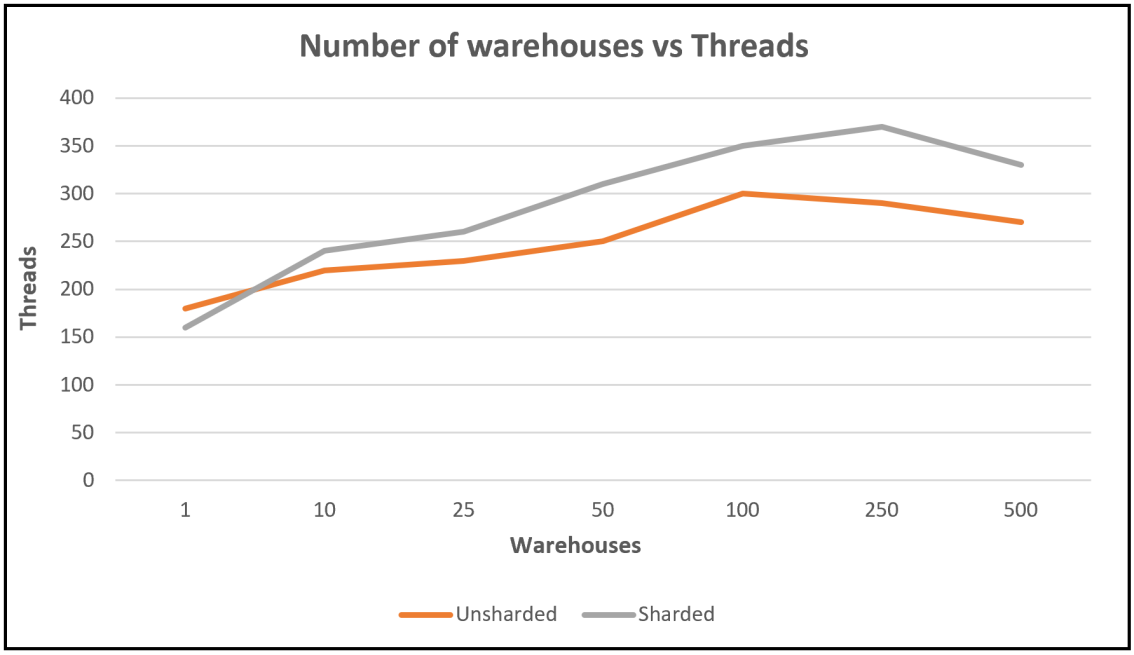


Figure 11: Number of warehouses vs. Threads for throughput saturation

Observation 1 - For a small data size (warehouse < 5), a sharded replica set exhibits less throughput as compared to an unsharded replica set for the same data.
Rationale – When sharding is used in system with less data, it becomes an overhead. Larger data size benefits from sharding because of more resources and data distribution therefore improves the throughput.

Observation 2 – Figure 10 and 11 shows that sharded replica set scaled throughput to 250 warehouses with 370 clients as compared to 100 warehouses of an unsharded replica set with 300 clients.
Rationale – Sharded replica set scales the system horizontally and thus allowing more concurrent clients to work on data as compared to an unshdarded replica set. The difference between the number of concurrent clients operating on the data in both cases is directly proportional to the relative throughput obtained.

Result Analysis - Results from experiment 3 suggest that MongoDB sharding can improve throughput, working with huge data while complying with ACID guarantees for the multi-document transactions.

7.5 Experiment 4 - FindAndModify vs find+update

Pytpcc provides a findAndModify configuration option that replaces a combination of "find and update" queries with a single "find and modify" query. FindAndModify updates the document that matches query criteria and gives a similar result as that of separate select and update queries [29]. In the new-order transaction, district update code can make use of findAndModify query, and we perform this experiment to check the impact of using findAndModify on the throughput. We used this configuration option for the referenced schema in an M40 cluster with 100 warehouses and executed the workload to see its impact on the throughput. Figure 12 shows the relative comparison between select+update and findAndModify.

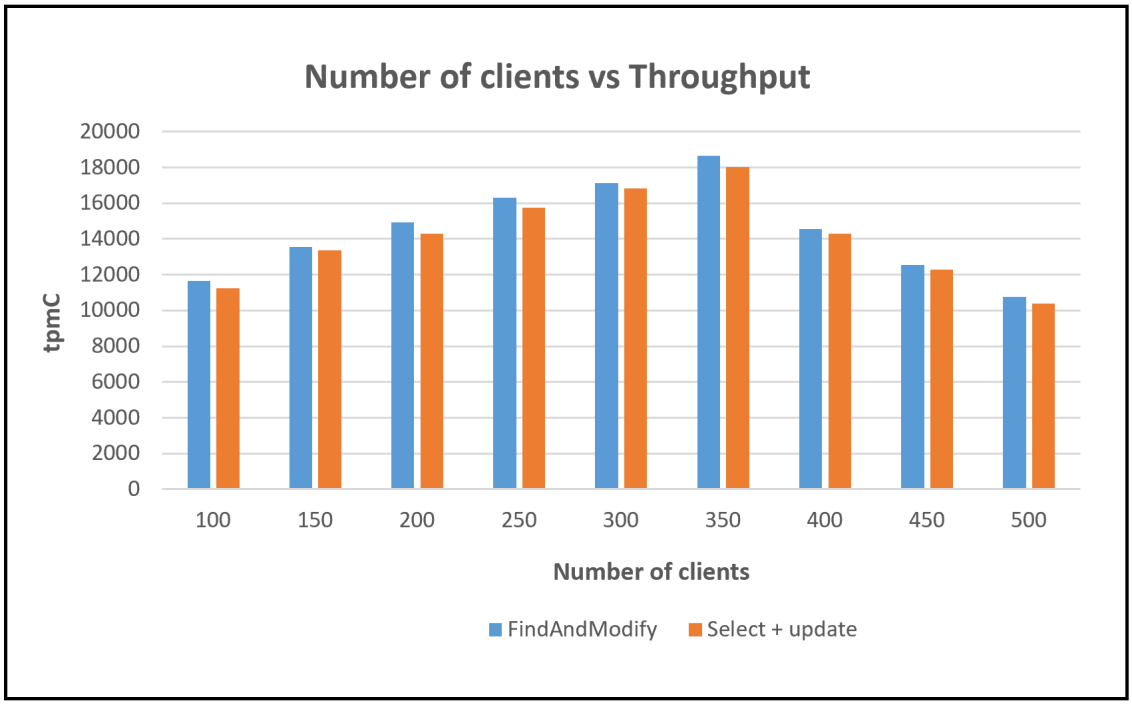


Figure 12: FindAndModify vs. find+update

Observation 1 – FindAndModify option increases throughput for a sharded cluster saturating with 350 clients.

Rationale – In new-order transactions, FindAndModify option reduces the number of queries for updating district information. The code snippet in Figure 13 shows the new-order transaction with FindAndModify configuration. It reduces the number of round trips to the database for the transaction, resulting in reduced latency and increased throughput.

```
if self.find_and_modify:
    d = self.district.find_one_and_update({"D_ID": d_id, "D_W_ID": w_id, "$comment": comment},
                                          {"$inc": {"D_NEXT_O_ID": 1}},
                                          projection=district_project,
                                          sort=[("NO_O_ID", 1)],
                                          session=s)
    if not d:
        d1 = self.district.find_one({"D_ID": d_id,
                                     "D_W_ID": w_id,
                                     "$comment": "new order did not find district"})
        print(d1, w_id, d_id, c_id, i_ids, i_w_ids, s_dist_col)
    assert d, "Couldn't find district in new order w_id %d d_id %d" % (w_id, d_id)

else:
    d = self.district.find_one({"D_ID": d_id, "D_W_ID": w_id, "$comment": comment},
                               district_project, session=s)
    assert d, "Couldn't find district in new order w_id %d d_id %d" % (w_id, d_id)
    # incrementNextOrderId
    d["$comment"] = comment
    self.district.update_one(d, {"$inc": {"D_NEXT_O_ID": 1}}, session=s)
```

Figure 13: FindAndModify new-order code

Result Analysis - Results from this experiment show us that MongoDB's powerful atomic updates like findAndModify() reduce the number of a round-trip to the database for a given query, which subsequently reduces individual operation latency in the multi-document transaction to boost the performance.

CHAPTER 8

Conclusion and future work

MongoDB multi-document transaction is a novel feature, and with ongoing developments, it has the potential to be used for business-critical scenarios. Our experiments in benchmarking multi-document transactions in the sharded cluster using industry-standard benchmark like TPC-C gave us valuable insights about read-write consistency, system scalability, and schema design choices for MongoDB.

The work our this research is wholly based on driver version 4.2 of the MongoDB community database. Use cases that require creating new collections, modifying system databases, and oplog of size more than 16 MB are currently excluded from MongoDB multi-document transactions. MongoDB is expected to add these features in its future driver version to encourage developers to use MongoDB transactions. This research can provide valuable guidelines to the NoSQL community and can be further extended with recent developments in the MongoDB driver.

LIST OF REFERENCES

[1] S. Choudhury, ‘‘Why are nosql databases becoming transactional?’’ Apr 2020. [Online]. Available: https://blog.yugabyte.com/nosql-databases-becoming-transactional-mongodb-dynamodb-faunadb-cosmosdb/

[2] ‘‘Read concern.’’ [Online]. Available: https://docs.mongodb.com/manual/reference/read-concern/

[3] ‘‘Write concern.’’ [Online]. Available: https://docs.mongodb.com/manual/reference/write-concern/

[4] ‘‘Overview of the tpc-c benchmark.’’ [Online]. Available: http://www.tpc.org/tpcc/detail5.asp

[5] S. Gilbert and N. Lynch, ‘‘Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services,’’ *ACM SIGACT News*, vol. 33, no. 2, p. 51, 2002.

[6] ‘‘Mongodb multi-document acid transactions,’’ *MongoDB*.

[7] A. Kamsky, ‘‘Adapting tpc-c benchmark to measure performance of multi-document transactions in mongodb,’’ *Proceedings of the VLDB Endowment*, vol. 12, no. 12, p. 2254–2262, 2019.

[8] ‘‘Couchbase brings distributed document acid transactions to nosql,’’ Jan 2020. [Online]. Available: https://blog.couchbase.com/couchbase-brings-distributed-multi-document-acid-transactions-to-nosql/

[9] M. Freels, ‘‘Achieving acid transactions in a globally distributed database,’’ Sep 2017. [Online]. Available: https://fauna.com/blog/acid-transactions-in-a-globally-distributed-database

[10] B. Calder, J. Wang, and A. Ogus, ‘‘Windows azure storage: a highly available cloud storage service with strong consistency.’’

[11] ‘‘G-store - a scalable data store for transactional multi key access in the cloud.’’ [Online]. Available: https://qfrd.pure.elsevier.com/en/publications/g-store-a-scalable-data-store-for-transactional-multi-key-access-

[12] J. Levandoski, D. Lomet, M. Mokbel, and K. Zhao, ‘‘Deuteronomy: Transaction support for cloud data.’’ 01 2011, pp. 123--133.

[13] D. Peng and F. Dabek, ‘‘Large-scale incremental processing using distributed transactions and notifications,’’ in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’10. USA: USENIX Association, 2010, p. 251–264.

[14] F. Junqueira, B. Reed, and M. Yabandeh, ‘‘Lock-free transactional support for large-scale storage systems,’’ *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2011.

[15] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, ‘‘Benchmarking cloud serving systems with ycsb,’’ *Proceedings of the 1st ACM symposium on Cloud computing - SoCC ’10*, 2010.

[16] A. Dey, A. Fekete, R. Nambiar, and U. Rohm, ‘‘Ycsb+t: Benchmarking web-scale transactional databases,’’ *2014 IEEE 30th International Conference on Data Engineering Workshops*, 2014.

[17] A. Dey, A. Fekete, and U. Röhm, ‘‘Scalable transactions across heterogeneous nosql key-value data stores,’’ *Proceedings of the VLDB Endowment*, vol. 6, p. 1434–1439, 2013.

[18] Mongodb-Labs, ‘‘mongodb-labs/py-tpcc.’’ [Online]. Available: https://github.com/mongodb-labs/py-tpcc

[19] W. Schultz, T. Avitabile, and A. Cabral, ‘‘Tunable consistency in mongodb,’’ *Proceedings of the VLDB Endowment*, vol. 12, no. 12, p. 2071–2081, 2019.

[20] M. Keep, ‘‘Mongodb multi-document acid transactions are ga: Mongodb blog,’’ Jun 2018. [Online]. Available: https://www.mongodb.com/blog/post/mongodb-multi-document-acid-transactions-general-availability

[21] D. Walker-Morgan, ‘‘Logical sessions in mongodb,’’ May 2019. [Online]. Available: https://www.mongodb.com/blog/post/transactions-background-part-2-logical-sessions-in-mongodb

[22] D. Walker-Morgan, ‘‘Local snapshot reads,’’ Jun 2019. [Online]. Available: https://www.mongodb.com/blog/post/transactions-background-part-3-local-snapshot-reads

[23] D. Walker-Morgan, ‘‘The global logical clock,’’ Jun 2019. [Online]. Available: https://www.mongodb.com/blog/post/transactions-background-part-4-the-global-logical-clock

[24] D. Walker-Morgan, ‘‘Low-level timestamps in mongodb,’’ May 2019. [Online]. Available: https://www.mongodb.com/blog/post/transactions-background-part-1-lowlevel-timestamps-in-mongodbwiredtiger

[25] D. Walker-Morgan, ‘‘Retryable writes,’’ Jun 2019. [Online]. Available: https://www.mongodb.com/blog/post/transactions-background-part-6-retryable-writes

[26] D. Walker-Morgan, ‘‘Safe secondary reads,’’ Jun 2019. [Online]. Available: https://www.mongodb.com/blog/post/transactions-background-part-5-safe-secondary-reads

[27] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux, ‘‘Oltp-bench,’’ *Proceedings of the VLDB Endowment*, vol. 7, no. 4, p. 277–288, 2013.

[28] Apavlo, ‘‘apavlo/py-tpcc,’’ Jan 2019. [Online]. Available: https://github.com/apavlo/py-tpcc

[29] ‘‘db.collection.findandmodify().’’ [Online]. Available: https://docs.mongodb.com/manual/reference/method/db.collection.findAndModify/

APPENDIX A

Referenced schema documents

```
CUSTOMER: {
    "C_ID": 1,
    "C_D_ID": 1,
    "C_W_ID": 1,
    "C_FIRST": "John",
    "C_MIDDLE": "Chris",
    "C_LAST": "Doe",
    "C_STREET_1": "2nd Market Street",
    "C_STREET_2": "1st San carlos",
    "C_CITY": "San Jose",
    "C_STATE": "CA",
    "C_ZIP": "17442",
    "C_PHONE": "1234567890",
    "C_SINCE": ISODate("2019-11-18"),
    "C_CREDIT": "GC",
    "C_CREDIT_LIM": 50000,
    "C_DISCOUNT": 0.4208,
    "C_BALANCE": 10000,
    "C_YTD_PAYMENT": 10,
    "C_PAYMENT_CNT": 1,
    "C_DELIVERY_CNT": 0,
    "C_DATA": "Test customer data",
}
```

```
ORDER: {
    "O_ID": 2452,
    "O_C_ID": 1,
    "O_D_ID": 1,
    "O_W_ID": 1,
    "O_ENTRY_D": ISODate("2019-11-18"),
    "O_CARRIER_ID": NumberLong(0),
    "O_OL_CNT": 13,
    "O_ALL_LOCAL": 1,
}
```

```
ORDER_LINE: {
    "OL_O_ID": 3675,
    "OL_D_ID": 1,
    "OL_W_ID": 1,
    "OL_NUMBER": 2452,
    "OL_I_ID": 67158,
    "OL_SUPPLY_W_ID": 1,
    "OL_DELIVERY_D": null,
    "OL_QUANTITY": 5,
    "OL_AMOUNT": 354.67,
    "OL_DIST_INFO": "Santa clara county"
}
```

```
HISTORY: {
    "H_C_ID": 2,
```

```
    "H_C_D_ID": 1,
    "H_C_W_ID": 1,
    "H_D_ID": 1,
    "H_W_ID": 1,
    "H_DATE": ISODate("2019-11-18"),
    "H_AMOUNT": 355.85,
    "H_DATA": "Sample history data"
}
```

APPENDIX B

Embedded schema documents

```
CUSTOMER: {
    "C_ID": 1,
    "C_D_ID": 1,
    "C_W_ID": 1,
    "C_FIRST": "John",
    "C_MIDDLE": "Chris",
    "C_LAST": "Doe",
    "C_STREET_1": "2nd Market Street",
    "C_STREET_2": "1st San carlos",
    "C_CITY": "San Jose",
    "C_STATE": "CA",
    "C_ZIP": "17442",
    "C_PHONE": "1234567890",
    "C_SINCE": ISODate("2019-11-18"),
    "C_CREDIT": "GC",
    "C_CREDIT_LIM": 50000,
    "C_DISCOUNT": 0.4208,
    "C_BALANCE": 10000,
    "C_YTD_PAYMENT": 10,
    "C_PAYMENT_CNT": 1,
    "C_DELIVERY_CNT": 0,
    "C_DATA": "Test customer data",
    "ORDERS":
    [
```

```
25      {
26              "O_ENTRY_D": ISODate("2019-11-18"),
27          "O_CARRIER_ID": NumberLong(0),
28          "O_OL_CNT": 13,
29          "O_ALL_LOCAL": 1,
30              "ORDER_LINE":
31              [
32          {
33                      "OL_NUMBER": 2452,
34                      "OL_I_ID": 67158,
35                      "OL_SUPPLY_W_ID": 1,
36                  "OL_DELIVERY_D": null,
37                  "OL_QUANTITY": 5,
38                  "OL_AMOUNT": 354.67,
39                      "OL_DIST_INFO": "Santa clara county"
40          }
41              ]
42      }
43      ]
44      HISTORY:
45      [
46      {
47          "H_D_ID": 1,
48          "H_W_ID": 1,
49          "H_DATE": ISODate("2019-11-18T07"),
```

```
        "H_AMOUNT": 355.85,
        "H_DATA": "Sample history data",
    }
    ]
}
```

APPENDIX C

Sample benchmark output

ubuntu@ip-172-31-7-168: /pytpcc: sudo python3 coordinator.py mongodb --no-load --duration 300 --warehouses 100 --clientprocs 200 –debug

```
{'debug': True, 'scalefactor': 1, 'warehouses': 100, '
   print_config': False, 'no_execute': False, 'clientprocs
   ': 200, 'no_load': True, 'stop_on_error': False, '
   config': None, 'system': 'mongodb', 'ddl': '/home/
   ubuntu/pytpcc/tpcc.sql', 'reset': False, 'duration': 60
   0}
```

```
{
  'ORDER_STATUS': 12492,
  'NEW_ORDER': 142620,
  'PAYMENT': 133307,
  'STOCK_LEVEL': 12175,
  'DELIVERY': 12512
}
```

```
{
  'total': 313106,
  'tpmc': 14262.362589899534,
  'write_concern': majority,
  'DELIVERY': {
    'total': 12512,
    'latency': {
```

```
      'p90': 337.73159980773926,
      'min': 117.76018142700195,
      'p75': 275.5575180053711,
      'p99': 1434.9379539489746,
      'p95': 396.0905075073242,
      'max': 6966.094732284546,
      'p50': 225.85463523864746
    },
    'retries_txn_total': 892
  },
  'batch_writes': True,
  'NEW_ORDER': {
    'total': 142620,
    'latency': {
      'p90': 151.6580581665039,
      'min': 15.34414291381836,
      'p75': 57.973384857177734,
      'p99': 680.8905601501465,
      'p95': 347.49531745910645,
      'max': 10055.912494659424,
      'p50': 36.026716232299805
    },
    'retries_txn_total': 18372
  },
  'ORDER_STATUS': {
```

```
    'total': 12492,
    'latency': {
      'p90': 21.37017250061035,
      'min': 7.789373397827148,
      'p75': 16.183137893676758,
      'p99': 77.51178741455078,
      'p95': 26.871681213378906,
      'max': 5207.825183868408,
      'p50': 13.137340545654297
    }
  },
  'causal': False,
  'aborts': 543,
  'read_concern': 'local',
  'threads': 200,
  'find_and_modify': False,
  'read_preference': 'primary',
  'warehouses': '10',
  'date': '2020-03-11 01:33:04',
  'denorm': False,
  'STOCK_LEVEL': {
    'total': 12175,
    'latency': {
      'p90': 17.210960388183594,
      'min': 7.087469100952148,
```

```
      'p75': 13.492584228515625,
      'p99': 35.06946563720703,
      'p95': 21.44336700439453,
      'max': 5094.253778457642,
      'p50': 11.312007904052734
    }
  },
  'retry_writes': False,
  'PAYMENT': {
    'total': 12512,
    'latency': {
      'p90': 2893.8026428222656,
      'min': 9.449005126953125,
      'p75': 1064.2321109771729,
      'p99': 10702.051639556885,
      'p95': 4633.890628814697,
      'max': 60109.302282333374,
      'p50': 149.30057525634766
    },
  },
  'all_in_one_txn': False,
  'txn': True,
  'duration': 600.01662611961365,
  'total_retries': 154770
}
```