Adding Transaction Query Processing for Non-Transactional NoSQL

Hiroshi HORII
IBM Research – Tokyo
NBF Toyosu Canal Front Building 5-6-52
Toyosu, Koto-ku Tokyo, 135-8511 Japan
+81-3-5144-2825
horii@jp.ibm.com

Tamiya ONODERA
IBM Research – Tokyo
NBF Toyosu Canal Front Building 5-6-52
Toyosu, Koto-ku Tokyo, 135-8511 Japan
+81-3-5144-2832
tonodera@jp.ibm.com

ABSTRACT

NoSQL supports high scalability with limited capabilities for transactions or query processing. Though emerging NoSQL, such as MongoDB, supports certain queries, and researchers are developing the techniques to enable transaction processing in the existing non-transactional NoSQL, there is no solution for both limitations. We are developing a new library, MongoTx, which enables transaction processing in MongoDB without loss of any query capabilities. MongoTx stores dirty data, which a client has modified but not yet committed, in MongoDB along with the committed data. Because the dirty data must not appear in the query results, MongoTx hides the dirty data from the queries. To efficiently process queries and transactions, MongoTx uses a special data model to store the data in MongoDB. With this data model, MongoTx handles the same queries as MongoDB while guaranteeing the atomicity, isolation, and consistency of the transactions. We evaluated MongoTx with YCSB and TPC-C, and show MongoTx can handle transactions without loss of the scalability of MongoDB.

1. INTRODUCTION

NoSQL databases, which support limited functions compared to traditional relational databases, are highly scalable and available due to their simplified architectures. Though they do not support the relational queries, transaction processing, and consistency of the traditional databases, their scalability and availability are providing benefits for modern Web services and even mission-critical services. However, as applications become larger and more complex, the features left out of NoSQL are sometimes needed. Therefore, new NoSQL systems, such as MongoDB[12] and Hive[14], support richer query languages. In addition, major Web companies, including Google and Yahoo, are developing techniques to support transaction processing on non-transactional databases[5][6].

Without modifying a non-transactional database, it can still support transaction processing if the dirty data and transaction states are stored in the database. Consider a client that inserts two records r_1 and r_2 into a table D, and then inserts a record r_t into a table T. If r_1 and r_2 include the primary key of r_t and the other clients can read r_1 or r_2 only after reading r_t, then the insertions of r_1 and r_2 seem to be atomic. Once the insertions of multiple records are atomic, then clients can build a versioning mechanism for the records by inserting records with the same key value and different version numbers. In addition, if clients read only the latest versions of the records in D, then the versioning of the records enables updates and deletions of records. Google Percolator[6] and Omid[5] use such techniques to process transactions with BigTable[4] and HBase[13].

Though the storage of dirty data and old versions enables transaction processing, this calls for complicated read-processing to guarantee the isolation and consistency of the transactions. Even if a client gets r_1 from D, that client can use r_1 only after checking r_t in T and any records following r_1 in D to prevent the use of dirty or stale records. Therefore, the existing solutions support only simple read operations, such as finding a record by its primary key by naively combining the native operations of their underlying NoSQL system. In general, rich query languages (such as MongoDB's) are not supported.

In this paper, we show how MongoTx enables transactions with rich queries in MongoDB without any additional middleware. MongoDB is a NoSQL database that stores and queries JSON documents, but lacks transactions. MongoTx works as a client library of MongoDB and allows MongoDB to process transactions with new three methods, `begin`, `commit` and `rollback`, which specify the scope of each transaction. MongoTx supports MongoDB queries while maintaining the isolation and consistency of the transactions by preventing them from using dirty or stale data. Like the other solutions that handle transactions in non-transactional NoSQL systems, MongoTx stores dirty data and transaction states in MongoDB by using its native client library. Because MongoTx does not require additional middleware, the replication mechanism of MongoDB remains available and improves the durability of the transactions.

To transactionally store and query JSON documents in MongoDB, MongoTx uses a new data model, *Single Document Two Versions* (*SD2V*). In SD2V, the database has at most two successive versions for each document by storing any newer version as a sub-document of the older one. For example, if a client replaces a document d_1 with document d_1', then d_1' is stored as d_1.*_unsafe* a sub-document of d_1. After commitment of the transaction, MongoTx replaces d_1 with d_1.*_unsafe*. Because it must be possible to commit d_1 at any time and d_1.*_unsafe* may not be committed at a particular time, we call d_1 the *safe version* and d_1.*_unsafe* the *unsafe version*.

SD2V provides two benefits for MongoTx in processing queries. First, MongoTx can easily identify the latest version because every fetched document always includes the latest version. Second, MongoTx can generate queries for unsafe versions by simply limiting the queries to safe versions, such as `{_unsafe.a: 100}`

generated with `{a: 100}` (which queries safe versions that have `100` in the field `a`).

With the SD2V data model, MongoTx supports the *read committed* and *repeatable read* isolation levels[2]. With these isolation levels, a client of MongoTx never reads an uncommitted (dirty) document as the result of any query. In addition, with the repeatable read, a client can read the same documents in the results of two queries within one transaction. MongoTx uses optimistic concurrency control to guarantee the isolation of these levels.

In addition, MongoTx guarantees clients will read the latest data, in the same consistency as the commercial databases[10][11] that support read committed and repeatable read. To process a query with these forms of consistency, MongoTx queries both the unsafe and safe documents in MongoDB, and merges the results into one result. Though the unsafe documents include dirty documents, MongoTx can identify the available documents that the clients can safely read.

The key contributions of this paper are:

- MongoTx, an efficient implementation that supports transactions with *read committed* and *repeatable read* isolation levels on top of MongoDB without regressions of sharding, indexing, replication and the aggregation framework functions of MongoDB.
- Single Document Two Versions (SD2V), a new data model that enables MongoTx to support MongoDB queries with atomicity, isolation, consistency of transaction.
- Empirical performance evaluation of MongoTx by using Yahoo! Cloud Serving Benchmarks and TPC-C-like benchmarks.

This paper is structured as follows: Section 2 describes background of MongoDB and transaction processing on NoSQL systems. Section 3 overviews the concepts of MongoTx and Section 4 and 5 describe the details of transactions and query processing respectively. Section 6 reports empirical performance results for MongoTx. Section 5 surveys the related work.

2. BACKGROUND

2.1 JSON

JSON is an open standard format to describe objects consisting of attribute-value pairs in the form of human-readable text, as shown in Figure 1. JSON was created for JavaScript to exchange data between servers and browsers. Many Web applications now store JSON objects in their databases. A JSON object consists of a collection of name and value pairs surrounded by braces ({ and }). Each name and value pair is associated with a colon (:) and pairs at the same level are separated by commas (,). The types of the values are number, string, Boolean, array, object and null, and each array can have multiple comma-separated values surrounded by square brackets ([and]). In this paper, we call a JSON object is a document, a JSON object within a JSON object is a sub-document, and a pair consisting of a name and a value is a field.

2.2 MongoDB

MongoDB is a database that stores documents by grouping them into *data collection*s in contrast to relational databases that store rows by grouping them into tables. Applications read and write documents in the data collections via the MongoDB APIs, which are supported in various programming languages.

2.2.1 Architecture

As with other NoSQL databases, MongoDB has functions for sharding, replication, and load balancing. Sharding is roughly equivalent to data partitioning in a relational database. Sharding splits all data into multiple partitions (shards) in the shared-nothing architecture[7]. A MongoDB daemon (`mongod`) manages each shard, and a router process (`mongos`) sends read and write requests to the appropriate `mongod`s with the proper shards. Multiple `mongod`s can be grouped to manage the same shard, and then the grouped `mongod`s replicate each update of the shard to improve the availability. MongoDB uses range-partitioning to split the documents and if a shard contains many more documents than the other shards, some ranges of the documents will be migrated to other shards and the routing information in all of the `mongos`es will be updated.

Concurrency is controlled in a `mongod` with a single readers-writer lock. While a `mongod` is possessing a write request, that `mongod` will not process any read requests. While a `mongod` is possessing requests to read one or more documents, that `mongod` will not process any write requests.

2.2.2 Basic API

MongoDB supports the basic CRUD operations, `insert`, `find`, `update`, and `remove`. In addition, MongoDB supports two test-and-set operations, `findAndModify` or `findAndRemove`, which select a document, then replace or remove the selected document while returning the document. MongoDB provides other useful functions, such as `findOne`, a special version of `find`, which returns only one document.

An ObjectId (corresponding to a primary key in relational database) is automatically or manually set for each document (as `_id` field) when the document is inserted. This ObjectId is unique in each data collection and each modification of an identified document is atomic.

MongoDB supports its own query language, which has many query operators to identify the documents to be written or read. The query operators are categorized as comparison (`$gt`, `$lt`, `$in`, etc.), logical (`$and`, `$or`, etc.), element (`$exists` and `$type`), evaluation (`$mod`, `$regex`, `$where`, etc.), geospatial (`$geoWithin`, `$near`, etc.) and array operators (`$all`, `$elemMatch`, and `$size`). Though queries across multiple collections (corresponding to JOIN operations) are not supported, clients can construct flexible queries with these operators (in particular `$where` allows clients to run custom JavaScript to select documents in `mongod`).

```
{  _id    :  1,
   name   : "Hiroshi",
   amount : 1000,
   friend : {  _id  : 10,
               name : "Tamiya" },
   group  : [
                {  name : "MongoDB"},
                {  name : "DB2"} ] }
```

Figure 1. An Example of a JSON document

```
1: DBCollection custs, hists;
 2:int payment(int whId, int custId, int amount) {
 3:  DBObject cust = custs.findOne({_id:custId});
 4:  cust["YTD_PAYMENT"] += amount;
 5:  custs.update({_id:custId}, cust);
 6:  DBObject hist = {
 7:   C_ID:custId, W_ID:whId, AMOUNT:amount };
 8:  hists.insert(hist);
 9:  int total = 0;
10:  DBCursor cursor =
11:    hists.find({W_ID: whId});
12:  while (cursor.hasNext())
13:    total += cursor.next()["AMOUNT"];
14:  return total;
15:}
```

Figure 2. Payment Procedure with MongoDB

Figure 2 shows a simplified payment transaction in TPC-C with a Java MongoDB API written using an extended Java grammar that handles the JSON documents as a primitive type of JavaScript. In Figure 2, each customer (`cust`) is identified with a customer ID (`custId`) via `findOne` in the customer data collection (`custs`) (Line 3) and the accumulated year-to-date payment (`cust["YTD_PAYMENT"]`) is updated via `update` (Lines 4-5). The payment amount, customer ID and warehouse ID (`whId`) are logged in the history data collection (`hists`) via `insert` (Lines 6-8) and the total of the payments to that warehouse is calculated using `find` is returned (Lines 10-13).

Because MongoDB does not support transaction processing to modify multiple documents, the insertion (Line 8) may fail after a successful update (Line 5) in Figure 2. Though this client may be able to roll back the update if the insertion fails, if the client fails while rolling back the update, then the update will never be rolled back. In addition, if multiple clients concurrently call the `update` at Line 5, then one client may destroy the other updates and the returned value of `payment` will be inconsistent with the total of the year-to-date values in `custs`.

2.2.3 Two phase commit on MongoDB

Performing two-phase commit with the native operations of MongoDB was described in the MongoDB user community[15][16]. Each client stores a log of the transaction operations in MongoDB before modifying any documents. The log has a state that starts as `init` and becomes `committed` after all of the modifications of the transaction are finished. The client generates a unique transaction ID and stores it as `_id` in the log. The client also stores this ID in a `tid` field for the all of the modified documents.

In the example of Figure 2 , before Line 2, the client generates a transaction ID (`myid`) and creates a document in a special collection (`txLogs`) as a transaction log:

```
{_id:myid, pay:[whId,custId,amount], state:"init"}
```

Next, the client sets `myid` as a field of the documents for the update of `cust`(Line 5) and the insertions of `hist` (Line 8):

```
cust["tid"] = myid;
hist["tid"] = myid;
```

After modifying and reading all of the documents (Line 14), the client changes the state of the transaction state:

```
{_id:myid,pay:[whId,custId,amount],state:"committed"}
```

Finally, the client deletes the stored `tid` fields from `cust` and `hist`, and then deletes the transaction state.

If the other clients obtain `cust` and `hist` documents that have `tid` fields, those documents are available only if the state of the transaction log for `myid` is `committed`. This means, the update of `cust` and the insertion of `hist` are atomically available to the other clients when the transaction state has changed to `committed`.

In addition, whenever the client fails while executing `payment`, the other clients can roll back or commit the updated `cust` and `hist` by analyzing the transaction log. For example, if the client fails *before* changing the transaction state to `committed`, then the other clients can rollback the updated `cust` by decreasing the `amount` in the transaction log and deleting the inserted `hist`.

Though storing transaction steps enables transaction processing on MongoDB as just described, special rollback logic is necessary for each transactional step. For this example, the rollback logic for the `payment` transaction is not applicable to the other transactions. It may be effectively impossible to implement logic for all of the possible transactions in a real application. Also, query processing is not feasible for documents modified by this technique.

2.3 Transaction Processing on NoSQL

To process general transactions on NoSQL without changing its implementation, some researchers have introduced new layers between clients and the NoSQL system. For example, Omid[5] provides an additional server to store transaction states, and Percolator[6] provides a client stub that communicates with BigTable[4]. Though the additional server reduces the durability of transactions because the stored transactions states must be persistent, the client stub approach remains durable by storing all of the transaction states in the underlying NoSQL.

Percolator is a client library that stores transaction states and dirty data in BigTable for general transaction processing. Percolator provides an API similar to a key-value store, storing each record in association with a key, and reading and writing its attributes by specifying the key and the attribute names.

All of the records are versioned by BigTable functions and the dirty data in a record is stored as a new version of the record. When a client modifies any attributes of a record in a transaction, the attribute becomes dirty, and Percolator adds a “lock” attribute into the record. While this attribute exists, Percolator allows only the modifying client to access the record.

When a client first modifies an attribute in a transaction, Percolator treats that attribute as the primary attribute of the transaction. Percolator appends links to the primary attribute to all of the other attributes that the client modifies in the same transaction. When a client wants to commit the transaction, Percolator erases the lock attribute of the primary attribute, and then erases all of the other lock attributes in the transaction.

Percolator recognizes a transaction is committed when the lock attribute of the primary attribute is erased. If a client fails after its Percolator has erased only the primary, but not all of the other lock attributes, some of lock attributes may remain even though its transaction is committed. In this case, Percolator on the other clients will continue erasing them by detecting their dead links to the primary attribute.

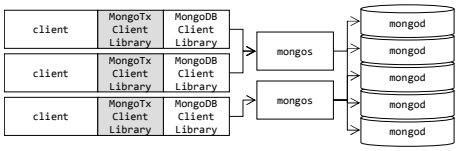

Figure 3. MongoTx Architecture

When a client reads an attribute of a record in a transaction, Percolator gets versions of the record and finds a version that has the latest attribute at the point of the start of the transaction. If the version does not exist, Percolator returns null, and if the version exists, Percolator checks its lock attribute. Only when lock attribute does not exist does Percolator return the value of the attribute, but otherwise Percolator rolls back the transaction and returns an error because some other client is updating it.

BigTable provides efficient implementation of versioning and Percolator uses it for versioning the records. In other words, it is difficult to apply the technique of Percolator efficiently on a NoSQL system that does not have such a versioning capability. For MongoDB, simulating versioning requires explicit querying and sorting of multiple documents for each read operation.

Percolator provides a Key-Value store API, with get and put for records, and can iterate, but does not support query processing. Therefore, to run TPC-E in the evaluation of Percolator, it was necessary to iterate all of the records in the tables of TPC-E. Iterating records increases the overhead and the lack of a query capability reduces the productivity of development.

3. MongoTx OVERVIEW

MongoTx is a new MongoDB client library that supports general transaction processing without any modification of MongoDB. MongoTx supports all of the MongoDB queries and the aggregation framework with API calls that correspond to the MongoDB API.

3.1 Architecture

MongoTx works as a client library of MongoDB as in Figure 3. Clients try to store documents with MongoTx and MongoTx stores them in the MongoDB servers with the native MongoDB client library. Because MongoTx works as just a client of MongoDB servers, replication, sharding, and indexing are available to increase its performance, scalability, and availability.

3.2 Atomicity

MongoTx has classes corresponding to the MongoDB classes: `TxSession` to `DB`, `TxDBCollection` to `DBCollection`, and `TxDBCursor` to `DBCursor`. `TxSession` has three additional methods, `begin` to start a transaction, and `commit` and `rollback` to finish the transaction. When a client calls `commit`, the documents modified since the last call of `begin` are atomically stored in MongoDB. In contrast, when a client calls `rollback`, the modified documents will be aborted.

Figure 4 shows the payment transaction of Figure 2 with the MongoTx API. The instance `session`, a `TxSession` instance, starts a transaction at Line 4 and finishes it at Line 16 by calling `begin` and `commit`. The instance `custs`, a `TxDBCollection` instance, queries and updates a document at Lines 5 and 7, and `hists`, another `TxDBCollection` instance, inserts one document in Line 10 and queries the documents at Lines 12-13.

```
1: TxSession session;
 2:TxDBCollection custs, hists;
 3:int payment(int whId, int custId, int amount) {
 4:  session.begin();
 5:  DBObject cust = custs.findOne({_id:custId});
 6:  cust["YTD_PAYMENT"] += amount;
 7:  custs.update({_id:custId}, cust);
 8:  DBObject hist = {
 9:   C_ID:custId, W_ID:whId, AMOUNT:amount };
10:  hists.insert(hist);
11:  int total = 0;
12:  TxDBCursor cursor =
13:    hists.find({W_ID:whId, C_ID:custId});
14:  while (cursor.hasNext())
15:    total += cursor.next()["AMOUNT"];
16:  session.commit();
17:  return total;
18:}
```

Figure 4. Payment Transaction with MongoTx

In the `commit` at Line 16, MongoTx atomically stores the documents modified since the last `begin` at Line 4. Therefore, if the updated document at Line 7 becomes available to the other clients, the inserted document at Line 10 is also available. When MongoTx cannot store these documents atomically, MongoTx rolls back the update and insertion, and then reports an error to the client.

3.3 Isolation

MongoTx supports *read committed* and *repeatable read* isolation levels. The default isolation level is read committed.

Both of the isolation levels prohibit *dirty write* and *dirty read*. If a client modifies a document, the document is unavailable to the other clients until its commit or rollback. In Figure 4, the updated and inserted documents at Lines 7 and 10 are unavailable to the others until the processing of `commit` at Line 16. In contrast, the result of the query in Lines 12-13 includes the inserted document from Line 10, because the client calls `insert` and `find` in the same transaction.

In addition, the isolation level of repeatable read prohibits *non-repeatable read*. Once a client reads a document as a query result, the client can read the same document in another query result in the same transaction. For the example of Figure 4, if the client calls `findOne` before calling `commit` to reload the document in the query result of Lines 12-13, then `findOne` must return the same document.

To enable these forms of isolation, MongoTx uses optimistic concurrency control: if MongoTx detects a violation of the rules of read committed or repeatable read, then MongoTx aborts the running transaction. Because MongoTx stores documents with a data model that is able to detect these violations, MongoTx can provide high performance for transaction processing.

3.4 Consistency

With MongoTx, a client can read the latest documents via `find` and `findOne`. In other words, MongoTx must process the `find` and `findOne` queries by reading all of the committed modifications when the client calls these methods.

As Percolator and Omidi, MongoTx stores dirty data of each transaction in MongoDB. In some cases, this data may remain dirty even after the commit of the transaction. If the dirty data is not reflected in the query results, then MongoTx cannot guarantee

consistency. Therefore, MongoTx detects dirty data that should be committed, and processes queries only after committing them. MongoTx uses an efficient data model to detect dirty data that should be committed and processes queries.

In addition, MongoTx provides a more relaxed consistency model to optimize the performance of queries. In this consistency model, the client specifies *t*-seconds with each query and MongoTx returns one of the latest query results in the past *t*-seconds. If the clients do not require strict consistency, then MongoTx provides better throughput.

3.5 Durability

MongoTx stores the data in MongoDB to maintain atomicity, isolation, and consistency of transactions. Therefore, the durability of transactions directly depends on the availability of MongoDB.

MongoTx can use most of the MongoDB functions that improve the availability of MongoDB. By replicating the updates of documents in multiple `mongod`s, MongoTx can tolerate failures as long as one of them survives. By using highly available disks (like RAID) as the storage for MongoDB, MongoTx rarely experiences `mongod` errors.

At the same time, MongoTx cannot use MongoDB configurations that permit loss of stored data. For example, MongoTx cannot accept replication settings that allow a `mongod` to take over as the primary for a sharding without synchronizing the documents stored in the previous primary.

3.6 Query

MongoTx supports MongoDB queries. Clients can use all of the operators to read documents with the guarantees of the ACID properties.

To process a MongoDB query, MongoTx does almost nothing to the query. A future release of MongoDB may extend its supported query operators or their semantics. To minimize future maintenance, MongoTx does not check the semantics of the query operators.

Instead of interpreting queries, MongoTx delegates most query processing to MongoDB. Because MongoTx uses a special data model to store documents, MongoTx modifies the read queries for the new data model. When a client tries to read a document with a query, MongoDB usually uses two or three queries. Because the modified queries are MongoDB queries, MongoTx can use the native client library of MongoDB to process them.

4. TRANSACTION PROCESSING

MongoTx stores dirty data and transaction states in MongoDB. To efficiently guarantee the isolation and consistency described in Section 3, MongoTx uses a new data model, Single Document Two Versions (SD2V) to store documents in MongoDB.

4.1 Transaction State

MongoTx stores the transaction state for each transaction in MongoDB. When MongoTx begins a transaction (`begin`), MongoTx stores that transaction's state into a special data collection, `txs`, and when MongoTx has finished all of processes of the transaction (with a `commit` or `rollback`), MongoTx removes the transaction state from `txs`.

```
{ _id    :  1,                                          safe fields
  Name   : "Hiroshi",
  amount : 1000,
  friend : { _id: 10, name: "Tamiya" },
  group  : [{ name: "MongoDB"}, { name: "DB2"} ],
  _unsafe: { _id:  1,                                   unsafe fields
             name: "Hiroshi",
             amount: 2000,
             friend: { _id: 10, name: "Tamiya" },
             group: [ { name: "MongoDB"},
                      { name: "DB2"} ] },
  _xTxId : "tx1",                                       metadata fields
  _sTxIds: [],
  _insert: false, _remove: false }
```

Figure 5. An Example of a SD2V document

As with the other documents, this transaction state is a JSON document that has a `state` field to represent its state, a `ts` field to remember the timestamp of the last modification of its `state`, and an `_id` field to identify the transaction ID. MongoTx generates a unique transaction ID for the `_id` field value when MongoTx inserts the transaction state document, and maintains a `ts` field value for each state transition.

The initial value of `state` starts with `"active"` and shifts to `"committed"` when MongoTx commits the transaction or `"rolledback"` when MongoTx rolls back the transaction. These value transitions are irreversible and MongoTx uses `findAndModify` to make the transaction transition to be deterministic.

When MongoTx successfully changes the `state` of a transaction state in `txs` to `"committed"`, then MongoTx knows the transaction is committed and makes the dirty data for the transaction available to all of the clients.

4.2 Single Document Two Versions

MongoTx uses a new data model, Single Document Two Version (SD2V) to store dirty and committed data into MongoDB. This data model enables MongoTx to insure the atomicity, isolation, and consistency of transactions.

SD2V is a data model for JSON documents and consists of three parts: safe fields, unsafe fields, and metadata fields. MongoDB stores all of the documents based on SD2V except the transaction state mentioned in Section 4.1. MongoTx maintains the SD2V documents for each transactional operation.

The safe fields store values that clients have already committed. After the commit of a transaction, MongoTx updates the safe fields with the values modified in the transaction. Because clients can avoid a *dirty read* by reading only the safe fields, we call these fields the *safe version*. However, the safe version may not be the latest. To keep the query results consistent, MongoTx needs to read the unsafe fields, the metadata and the transaction states in the MongoDB.

The unsafe fields are in a sub-document as a special field `_unsafe`. We call this sub-document is the *unsafe document*. When a client updates a document, MongoTx creates an unsafe document, copies the safe fields to the unsafe fields, and then modifies the unsafe fields. Therefore, unsafe fields always have the same values as safe fields except for any updated fields. After MongoTx commits a transaction, MongoTx replaces the safe fields with the unsafe fields, and then deletes the unsafe document. We call

unsafe fields an *unsafe version* because they may not be committed.

The metadata fields consist of `_xTxId`, which represents the transaction with the exclusive lock to access the document, `_sTxIds`, which represents the transactions with the shared lock to access the document, and `_insert` and `_remove`, which represent the unsafe documents for insertions or removals. The value of `_xTxId` is a transaction ID and the value of `_sTxIds` is an array of transaction IDs.

Using `_xTxId` and `_sTxIds`, MongoTx insures the isolation of transactions. When a client reads a document with repeatable read, MongoTx appends the client's transaction ID to `_sTxIds`. When a client modifies a document, MongoTx modifies `_xTxId` with the transaction ID. MongoTx can modify these fields only when the `_xTxId` of the document does not have a value. MongoTx uses `findAndModify` to atomically check the value of `_xTxId` and to modify the fields.

MongoTx sets the `_insert` field of a document to `true` when a client inserts the document. In addition, MongoTx sets the `_remove` field to `true` if any clients remove the document. Therefore, there are no safe fields when the `_insert` field is `true`, and there are no unsafe fields if the `_remove` field is `true`.

The document of Figure 5 shows an example of an SD2V document when the `amount` field of Figure 1 has been updated from `1000` to `2000`. The safe and unsafe fields of this document are the same as in Figure 1 except for the updated `amount` field in the unsafe fields. The `_xTxId` field indicates that the client that generated a transaction ID `tx1` updated the `amount` field.

MongoTx uses `findAndModify` or `findAndRemove` to modify the SD2V fields of each document. Therefore, only one client can modify the SD2V fields at one time even when multiple clients simultaneously try to modify the document. If a modification fails, then MongoTx checks the latest SD2V fields, and then retries the modification or rolls back a transaction if MongoTx detects violations of isolation or consistency.

4.3 Transaction Processing with SD2V

MongoTx enables transactions by modifying documents in `txs` and SD2V documents in data collections. Figure 6 shows the changes in the documents in the data collections from Figure 4. The first column shows documents in `txs`, and the second and third column shows documents in `custs` and `hists`. There is one document in `custs` and two documents exist in `hists` before the start of the `payment` transaction.

First, when a client starts a transaction at Line 4, MongoTx generates a new transaction ID (`"tx1"`), obtains the current clock value (`080520140101`), and then stores them in `txs` as an active transaction state as shown in Figure 6-A.

Next, when a client tries to read a document in `custs` at Line 5, MongoTx finds a document with its query processing, and then stores `tx1` into its `_sTxIds` field. When a client updates a document with a new document in `custs` at Line 7, MongoTx gets its latest document from the MongoDB, and then sets the new document and `tx1` to the `_unsafe` field and the `_xTxId` field of the document, respectively. Figure 6-B shows the documents in the MongoDB after these steps.

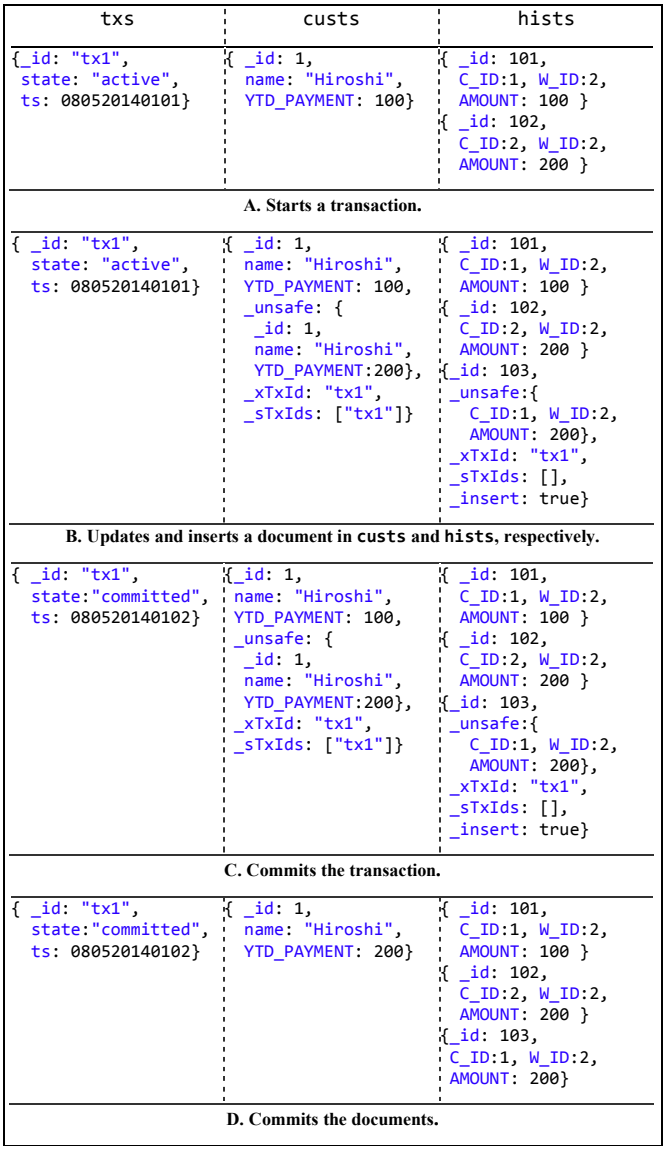

Figure 6. Payment processing with SD2V documents

When a client commits a transaction at Line 16, MongoTx changes the `state` field of its transaction state in `txs` from `"active"` to `"committed"`, as shown in Figure 6-C. Next, for each document modified in the transaction, MongoTx replaces the safe fields with the unsafe fields, deletes the unsafe documents, and then clears the metadata changes (`_insert`, `_xTxId`, and `_sTxIds`), as shown in Figure 6-D. Finally, MongoTx removes the transaction state from `txs`.

When a client rolls back a transaction, MongoTx changes its transaction state from `"active"` to `"rolledback"`, and then deletes all of the unsafe documents and the metadata modified in the transaction.

4.4 Isolation Support

MongoTx prevents its clients from dirty write and dirty read for *read committed*, and protects against dirty write, dirty read, and

non-repeatable read for *repeatable read*[2] by using optimistic concurrency control.

Dirty write: When a client c_1 modifies a document from d_1 to d_2 and then another client c_2 modifies it to d_3, it is unclear what the correct document should be if c_1 rolls back their modifications of the document. To protect against this situation, MongoTx does not allow a client to modify a document until the other client commits or rolls back its modification of the document. In the SD2V model, MongoTx can identify the transaction that last modified any documents by referring to its `_xTxId` value. If the transaction identified by the `_xTxId` is not yet committed nor rolled back yet, then MongoTx throws an error to the client that tries to modify the document.

Dirty read: When a client c_1 modifies a document from d_1 to d_2, and then another client c_2 reads d_2, the c_2's read is inconsistent if c_1 rolls back the modification. To protect against this situation, MongoTx allows a client to read only the safe fields of documents except for the unsafe fields modified by itself. In the SD2V model, MongoTx applies the modifications in a transaction to the safe fields after committing the transaction. By reading only the safe fields of documents, MongoTx can avoid any dirty read. However, a client needs to read the unsafe fields modified by itself. Therefore, if the `_xTxId` field of a document is the client's own transaction ID, MongoTx makes its unsafe fields available.

Non-repeatable read: If a client c_1 modifies a document from d_1 to d_2 after the client c_2 reads d_1, then c_2 cannot reread d_1 after the modification. To protect against this situation, MongoTx does not allow a client to modify a document while another client might be reading the document in its own transaction. In transaction processing with SD2V, MongoTx stores a transaction ID in `_sTxIds` before a client reads the document, and removes the transaction ID after the client finishes its transaction. Therefore, if the `_sTxIds` contains at least one transaction ID, then the client may reread the document again because that client has not finished its transaction. To return the same document to the client, MongoTx prohibits the other clients from modifying the document while the `_sTxIds` field has transaction IDs.

5. QUERY PROCESSING

If a document has an unsafe document, then the unsafe document may be in a committed state. Therefore, if a client queries the latest documents, MongoTx needs to evaluate both the safe and unsafe fields. However, the unsafe fields may not be committed, and to avoid a dirty read a client must not read the unsafe fields. Therefore, MongoTx must know if the transaction is committed before using the unsafe fields. In addition, a client that created unsafe fields may fail. When using unsafe fields, MongoTx analyzes any transactions started by a failed client.

5.1 Repair Documents

A client may start a transaction and modify a document, but then fail without the commit of the transaction. Even when a client commits the transaction by changing its transaction state, the client may fail without replacing the safe fields with the unsafe fields. In Figure 6, if the client of `tx1` fails between Figure 6-B and Figure 6-C, the other clients cannot modify the document in `cust` to avoid dirty writes. If the client fails between Figure 6-C and Figure 6-D, the other clients cannot read the latest document in `cust`.

```
1: DBCollection c;
 2:DBObject repair(DBObject d) {
 3:   if(d==null || d["_xTxId"]==null) return d;
 4:   Object txId = d["_xTxId"];
 5:   DBObject s = txs.find({_id:txId});
 6:   if(s == null) { ; //nop
 7:   } else if(s["state"]=="committed") {
 8:     if(d["_remove"])
 9:       c.findAndRemove({_id:d["_id"],_xTxId:txId});
10:     else
11:       c.findAndModify(
12:       {_id:d["_id"],_xTxId:txId}, d["_unsafe"]);
13:   } else if(s["state"]=="rolledback") {
14:     d["_unsafe"]=null;
15:     d["_xTxId"]=null; d["_sTxIds"].remove(txId);
16:     if(d["_insert"] == null) {
17:       d["_insert"] = null;
18:       c.findAndRemove({_id:d["_id"],_xTxId:txId});
19:     } else {
20:       d["_remove"] = null;
21:       c.findAndModify(
22:        {_id:d["_id"],_xTxId:txId}, d);
23:   } else if(s["ts"]+timeout > now()) {
24:     s["state"]="rolledback"; s["ts"] = now();
25:     txs.findAndModify(
26:       {_id:d["_xTxId"], state:"active"}, s);
27:   } else { error(); }
28:   return repair(c.find({_id:d["_id"]}));
29:}
```

Figure 7. Repair Algorithm

When MongoTx processes a query, MongoTx repairs the safe fields of the documents in MongoDB to insure they are the latest values: MongoTx replaces the safe fields with the unsafe fields if they have been committed. While repairing documents, MongoTx detects client failures and rolls back documents modified by failed clients. Figure 7 shows the algorithm to repair a document in a data collection `c`.

When a client reads and modifies a document that has an unsafe document, MongoTx identifies the transaction with the `_xTxId` field of the document (Line 5), and checks its status in `txs` (Lines 6-27). If the transaction has already been committed, then MongoTx applies the modifications in the unsafe document to the safe fields (Lines 8-12). If the transaction has already been rolled back, then MongoTx removes the modifications in the unsafe document from the document (Lines 14-22). If the transaction has been active for more than a set time (*timeout*), MongoTx rolls the transaction back by changing the state in `txs` (Lines 24-26).

5.2 Evaluating Query

To process a query, MongoTx first evaluates the unsafe fields of the documents with the repair function of Figure 7. Next, MongoTx evaluates the safe fields of the documents. Figure 8 shows the algorithm to find documents with a query `q` in a data collection `c` with a transaction ID `txId`.

5.2.1 Evaluating Unsafe Fields

When a client tries to read documents with a query, MongoTx repairs the documents in MongoDB. The most naïve way to repair documents would be if MongoTx checked all of the documents in MongoDB with the repair function of Figure 7. However, typical queries only read a few documents. By selecting the documents to

be repaired for each query, MongoTx reduces the overhead to process queries.

By using the same query that the client sent, MongoTx generates a query to select documents possibly needing repairs. The algorithm to generate the query is simple: MongoTx just prepends the prefix, "`_unsafe.`" before all of the top-level fields (Lines 4-5)[1]. For the query in Line 13 of Figure 4, MongoTx generates this query.

```
{ _unsafe.W_ID: whId, _unsafe.C_ID: custId }      :Q1
```

For the results of the `find` method with Q1, MongoTx uses the repair algorithm from Section 5.1 to update them (Lines 6-12). The results may include unsafe documents generated in the same transaction. For example, MongoDB in Figure 6-C will return a document `{_id: 103,…}` from `hists` with that query. Because these unsafe documents must be available to the client that generated the document, MongoTx uses the document as a part of the query result (Lines 9-10). For the other documents in the query results, MongoTx uses the repair algorithm to move all of the unsafe fields to the safe fields (Line 11).

After MongoTx uses the repair algorithm, MongoDB may store unsafe documents that match the condition of the query. These unsafe documents were generated by the other clients after the client started to read. Because MongoTx guarantees the query result includes all of the committed documents before the client tries to read, MongoTx ignores these unsafe documents.

```
1: Object txId; DBCollection c;
2:TxDBCursor find(DBObject q) {
3:   List<DBObject> ret = new ArrayList<>;
4:   DBObject uq = [generate a query from q
5:                  by prepending "_unsafe" to q's fields];
6:   DBCurosr cur = c.find(uq);
7:   while(cur.hasNext()) {
8:     DBObject d = cur.next();
9:     if(txId.equals(d["_xTxId"]))
10:      ret.add(doc["_unsafe"]);
11:    else repair(d);
12:  }
13:  List<DBObject> repaired = new ArrayList<>();
14:  cur = c.find(q);
15:  while(cur.hasNext()) {
16:    DBObject d = cur.next();
17:    if(d["_xTxId"] == null) ret.add(d);
18:    else if(txId.equals(d["_xTxId"])) continue;
19:    else {
20:      DBObject rDoc = repair(d);
21:      if([rDoc's and d's committed fields are the same)
22:        ret.add(d);
23:      else repaird.add(rDoc["_id"]);
24:  }}
25:  DBObject rq = [generates a query from q
26:                 by adding an $in closure with repaired];
27:  cur = c.find(rq);
28:  while(cur.hasNext()) {
29:    DBObject d = cur.next();
30:    if (d["_xTxId"] == null); ret.add(d);
31:  }
32:  return [generate a TxDBCursor from ret];
32:}
```

Figure 8. Query Algorithm

[1] In case of `$where`, MongoTx replaces `this` and `obj` with `this._unsafe` and `obj._unsafe` respectively in the JavaScript.

5.2.2 Evaluating Safe Fields

After evaluating the unsafe fields, MongoTx finds the documents that have safe fields matching the conditions of the client query. Because the safe fields are the same as the fields used in the client query, MongoTx can use the client query itself to find the documents (Lines 13-20).

The results of the client query may include unsafe documents. Consider this client request to find documents that have a value greater than `10` in the `YTD_PAYMENT` field.

```
{ YTD_PAYMENT: {$gt: 10}}                          :Q2
```

The document in `cust` in Figure 6-C matches this condition. However, because the latest version of this document is its unsafe document, MongoTx must not use the document in `cust` as the result. Therefore, if MongoTx finds the unsafe document of a document in the results, it repairs the document (Line 20). After repairing the document, if no change exists in the committed fields of the document, MongoTx uses the document in the query result (Lines 21-22). In addition, MongoTx can immediately use any documents in the query result if there are no unsafe documents (Line 17).

If the committed fields are changed in the repair algorithm, then the new committed fields may still match the query condition. For the example of Q2 in Figure 6-C, though the document in `cust` will be repaired, the repaired document still matches the condition of Q2. Therefore, MongoTx remembers the repaired documents (Line 23), and again searches the repaired documents (Lines 25-27). For the current example, because MongoTx repaired the document `{_id: 1, … }`, MongoTx will send this query:

```
{ YTD_PAYMENT: {$gt:10}, _id: {$in:[1]} }          :Q3
```

MongoTx uses the returned documents in the query results if they do not have any unsafe documents (Lines 28-31).

5.3 Aggregation support

By using the aggregation functions, the clients of MongoDB can find documents in a data collection with complex query operations, with sorts, group-by, and the other options. For example, a client can get a list of documents that have `whId` as their `W_ID` field sorted by `W_ID` in descending order.

```
custs.aggregate(
  { $match: { W_ID: whId} },
  { $sort: { W_ID: -1} }
);
```

MongoTx supports the aggregation functions when clients use read committed. Before processing the aggregation operators, MongoTx repairs all of the unsafe documents in the data collection (MongoTx ignores the error in Line 27 in Figure 7 while processing aggregation functions). Then MongoTx modifies the aggregation operators by inserting a new `$project` operator before the first aggregation operator, and converts the client's operators, producing this query:

```
custs.aggregate(
  { $project:
    { _merged: { $cond:
       { if: { $eq: [ "$_xTxId", txId]},
         then: "$_unsafe",
         else: "$$CURRENT"
  }}}},
  { $match: { _merged.W_ID: whId} },
  { $sort: { _merged.W_ID: -1} }
);
```

The inserted `$project` operator transforms each document depending on the `_xTxId` field value. If the `_xTxId` field value is the same as the transaction ID of the client (when `_xTxId` is `txId`), then MongoDB moves the unsafe document to the `_merged` subdocument, but otherwise MongoDB moves the root document (`$$CURRENT`) to the `_merged` subdocument.

MongoTx converts the `$match` and `$sort` aggregation operators in the same way as used in the evaluation of unsafe fields. Instead of `"_unsafe."` in Section 5.2, MongoTx adds the `"_merged."` prefix before the top field name for the `$match` and `$sort` aggregation operators.

Finally, MongoTx calls the `aggregate` method, obtains a list of `_merged` subdocuments from MongoDB, and returns the list to the client.

5.4 Optimization

5.4.1 Fetching a document with `_id`

MongoDB prohibits any modification of the `_id` field in the stored documents. Therefore, even if a document has an unsafe document, the `_id` field values in the committed fields and the unsafe fields must be the same.

If a client tries to read a document with specifying only an `_id` field value, MongoTx skips the evaluation of the unsafe fields in Figure 8 (in particular, MongoTx starts the execution of the query algorithm from Line 13). Because the `_id` field is specialized for the performance in MongoDB, MongoTx can fetch the document effectively by using only `_id` field, but not `_unsafe._id` field.

5.4.2 Read Only Transaction

MongoTx creates a transaction state in MongoDB when a client starts a transaction, and deletes the transaction state when the client commits the transaction as described in Section 4.1. If the transaction is read-only and the client uses read committed, then MongoTx does not modify SD2V documents when running the transaction. Because the other clients can find any transaction states only through SD2V models, they never read the transaction state.

MongoTx lazily creates a transaction state if a client uses read committed. When the client requests the first write operation in a transaction, MongoTx creates the transaction state. Therefore, if the client requests a read-only transaction, MongoTx does not create a transaction state in MongoDB.

5.4.3 Reading Stale Documents

In the repair algorithm in Figure 7, MongoTx rolls a transaction back if the transaction reads a document that another active transaction is updating (Line 27). However, in most cases, the safe fields of the document were the latest data *`timeout`* seconds earlier because transactions must finish within the time.

By allowing a transaction to read stale but committed fields, MongoTx can avoid rolling the transaction back. This optimization is enabled by returning `d` in Line 27 of Figure 7 (after removing the unsafe document and the metadata fields). With this optimization, if a client specifies *t*-second *`timeout`* value, then MongoTx returns query results that may includes documents with the staleness bounded within the past *t* seconds.

5.4.4 Indexing Unsafe Fields

MongoDB provides an indexing mechanism for better query performance. If a client adds an index to one or more fields in a data collection, then each `mongod` generates a B-tree for the index of those fields. MongoDB uses the indexes while processing a query if the query has conditions for those fields.

As described in Section 5.2.1, MongoDB evaluates unsafe fields in the same way as committed fields when processing queries. Therefore, by adding the indexes to the unsafe fields in the same way as the safe fields, MongoDB can process queries more efficiently. Therefore, if a client adds an index to the committed fields, then MongoTx adds the same index to the unsafe fields.

5.4.5 Sharding

Sharding adds the high scalability to MongoDB. To enable sharding for a data collection, some of the fields in the data collection are configured as a shard key. MongoDB prohibits any modification of the fields of the shard key. In addition, every document in the data collection must have the field values as the shard key.

By extending the SD2V model, MongoTx supports the sharding. In the SD2V model, when a document is inserted in a transaction, it is stored as an unsafe document of a document without any committed fields in MongoDB. This means, no field of the shard key exists in the inserted document. Therefore, if sharding is enabled for a data collection, MongoTx copies the unsafe fields of the shard key to the committed fields.

5.4.6 Supports MongoDB APIs

MongoTx provides the same APIs with MongoDB. MongoTx has the classes and methods that correspond to MongoDB APIs. MongoTx handles each read or write method in MongoDB APIs as a transaction with single method that corresponds to the method. When a client calls a method of MongoDB API, MongoTx starts a transaction, calls the corresponding method, and then commits the transaction.

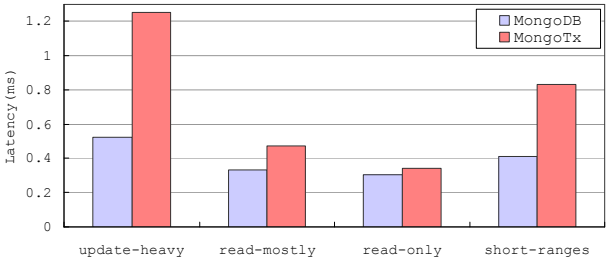

Figure 9. Latency of YCSB (Single Client, Single Node)

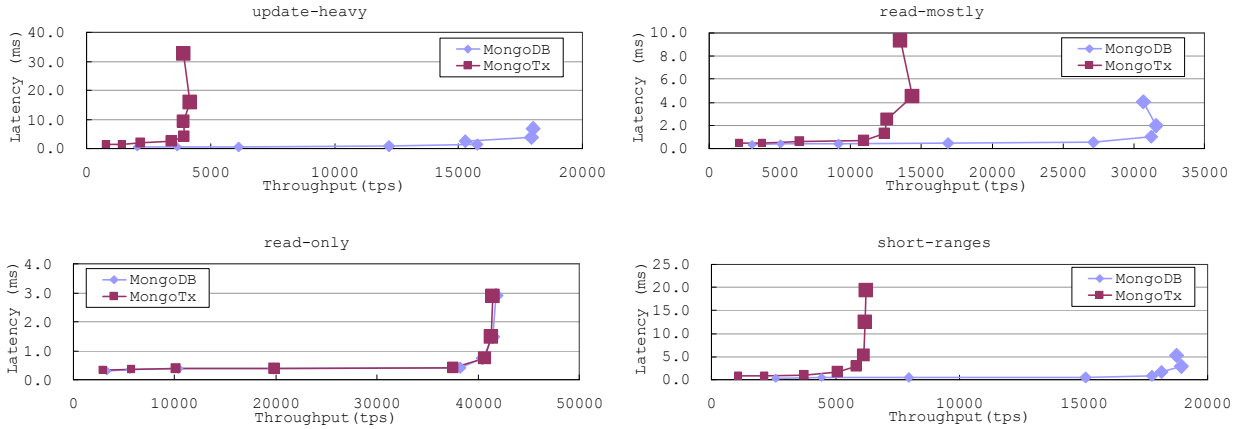

Figure 10. Throughputs and Latencies of YCSB (Single Node)

SD2V data model is compatible with data model that native MongoDB application uses. The committed fields are the same as the fields of the application and the other fields of SD2V are optional when the application starts using MongoTx's transactions. Therefore, without any migration of documents in MongoDB, the application can enable transactions only by replacing the original MongoDB library with the MongoTx library.

6. EVALUATION

In this section, we evaluate the performance of transactions and queries in MongoTx. First, we compared the basic performance of the original MongoDB and MongoTx by using YCSB (Yahoo! Cloud Serving Benchmark). Second, we evaluated a complex benchmark based on the TPC-C benchmark. For these evaluations, we used five `mongod` machines, three `mongos` machines, and one client machine, each with one Intel(R) Xeon(R) processor E5-2680(2.70GHz, eight cores, hyper-threading enabled) and one SAS hard disk drive. For every evaluation, we measured five times the averages of the throughputs and latencies.

6.1 Microbenchmarks

We determined the cost of the transactions and queries as handled by MongoTx with YCSB. YCSB is intended to benchmark data storage systems for cloud-based services by providing configurable workloads that simulate Web applications. There are six predefined core workloads with YCSB: update-heavy, read-mostly, read-only, read-latest, short-ranges, and read-modify-write. We used update-heavy, read-mostly, read-only, and short-ranges with the respective read-write ratios of 50:50, 95:5, 100:0 and 95:5. One document is updated or fetched in each YCSB operation except for the read operation of short-ranges. For a read operation in short-ranges, ten documents are read by using a query. We used a zipfian distribution for the clients to access the values of the keys and configured the number of the records to 100,000.

Because the original implementation does not support transactions, we modified YCSB to add support for transactions. Each transaction executes five YCSB operations with the read committed isolation level, and then commits the updated documents. In this evaluation, we measured the throughputs based on the completed YCSB operations.

First, we evaluated the overhead of MongoTx with one machine. Figure 9 shows the latencies of YCSB when one client ran. Because MongoTx always processes one transaction, rollback and repair never happen in this evaluation. In Figure 9, the latency of MongoTx in read-only showed the mostly similar performance to MongoDB. However, the latency of MongoTx in the other workloads was worse than MongoDB. In addition, when the write ratio in workloads increased, the overhead became worse.

Figure 10 shows the throughputs and latencies of YCSB when we used 1, 2, 4, 8, 12, 32, and 128 clients (the size of plots depends on the number of clients). In update-heavy, read-mostly, and short-ranges, the peak throughputs of MongoTx were worse than MongoDB. When the write ratio in workloads increased, the overhead of MongoTx became worse.

Figure 11 shows the averaged number of MongoDB operations to process one YCSB operation. To enable transaction, MongoTx calls more APIs than MongoDB. The characteristics of Figure 11 are similar to Figure 9. This means, the additional API calls was the major reason to increase the latency of MongoTx.

Next, we evaluated the throughputs of YCSB with multiple machines. Figure 12 shows the throughputs normalized with the throughput of the single machine. The scalability of MongoTx was equals or better than MongoDB. This means, MongoTx inherits MongoDB scalability though MongoTx has overhead of transactions and queries in the YCSB workloads.

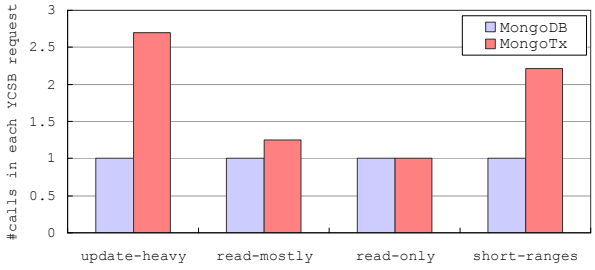

Figure 11. Numbers of MongoDB API calls for each YCSB Operation (Single Node)

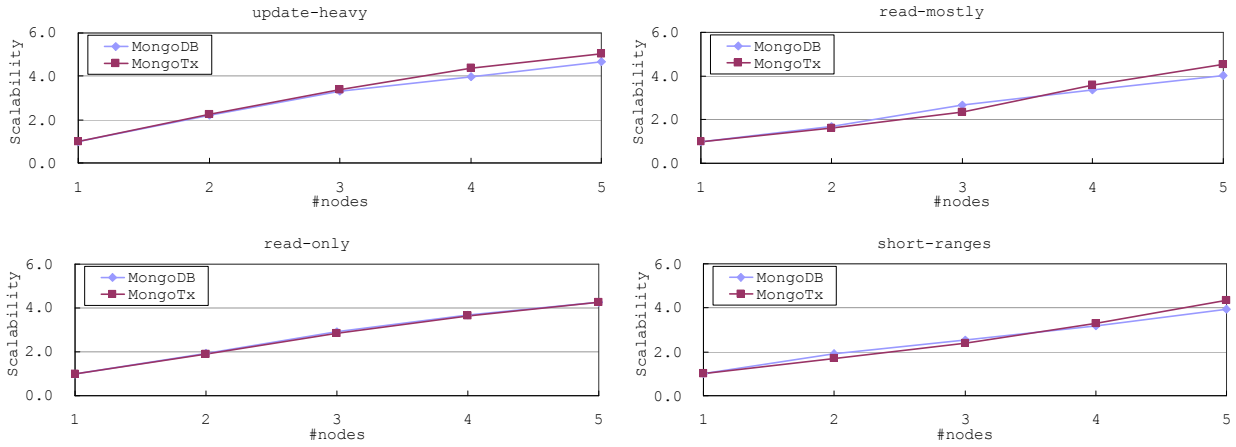

Figure 12. Scalability of YCSB

6.2 Synthetic Workload

To evaluate MongoTx with a more realistic scenario, we implemented benchmarks based on TPC-C[9]. TPC-C is designed for OLTP systems with five types of transactions: payment, order status, delivery, stock level, and new order. The order status and stock level are read-only transactions and the others are transactions with updates. We have implemented two benchmarks based on the TPC-C benchmark with the MongoDB and MongoTx APIs. In the benchmarks, a client queries documents once in two transactions. Because MongoDB has no transaction, the benchmark with MongoDB APIs supports only single client.

First, we evaluated the latencies of two benchmarks when one client runs. In this evaluation, MongoTx does not process any rollback and repair. Therefore, we can determine the overhead of transaction and query of MongoTx in the ideal situation.

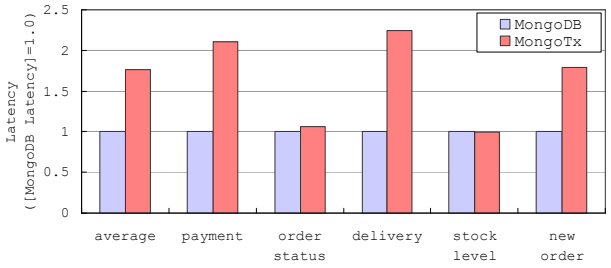

Figure 13. Latency of TPC-C-like benchmarks (Single Client)

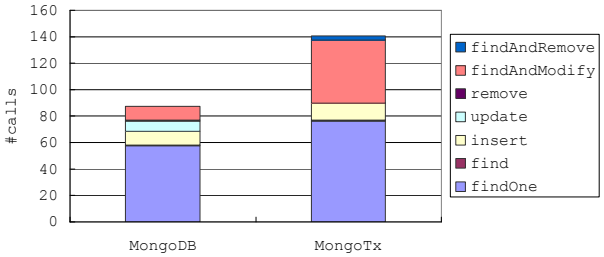

Figure 14. Distribution of API calls in TPC-C-like benchmarks (Single Client, Single Node)

Figure 13 shows the latencies of average and each transaction when we used one client and one `mongod` server. As the YCSB evaluation, the latencies of MongoTx in read-only transactions showed the mostly similar performance to MongoDB. In addition, the averaged latency was only 70% worse even though MongoTx pays overhead of transactions and queries.

Figure 14 shows the averaged distributions of API calls for one transaction in two benchmarks. The relative difference between the total numbers of the API calls was mostly similar to the differences of the averaged latencies in Figure 13. The major difference in two distributions was the number of `findAndModify`. MongoTx uses it to change transaction states, adds unsafe documents, and commits them. Therefore, the transaction processing in MongoTx causes the main overhead in the benchmark.

Finally, we evaluated the scalability of the benchmark of MongoTx. Figure 15 shows the relative throughput based on the single server throughput. The throughput improved most-linearly based on the number of the `mongod` servers. With this result, we can determine the overhead of transactions and query did not matter for the scalability of MongoDB in the benchmark.

Recently, the computing resources are becoming cheaper by the benefits of the cloud technologies. We believe the high scalability of MongoTx with transactions and queries provide more value than their overhead.

7. RELATED WORK

Percolator[6] takes the same approach with MongoTx to add transactions on top of BigTable[4]. Percolator works as the proxy of BigTable and keeps the isolation of transactions with locks and optimistic concurrency control. Percolator supports only Key-Value store API and does not support query languages.

Omid[5] needs additional servers to add transactions on top of Hbase. A client stores a transaction state in the servers and stores dirty data in Hbase[13]. The dirty data is available only after the transaction state becomes in a committed state. Because the additional servers work on the physically different machines from the Hbase servers, additional considerations are necessary to keep the availability. In addition, Omid does not support query.

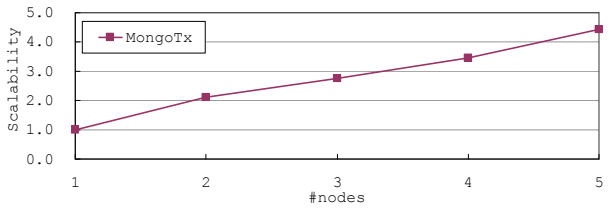

Figure 15. Scalability of a TPC-C-like benchmark

Google Megastore[1], which is used in Google's major services such as Gmail, Picasa, Calendar, manages entity groups and supports atomic updates of records in multiple entity groups with two-phase commit. Megastore uses the middleware servers between clients and data storage to manage transaction states and replicate updates. BigTable uses BigTable as the data storage and the data model is hierarchical. Megastore clients can query hierarchical data by using their indexing. However, because Megastore provides the different APIs from BigTable, the application of BigTable must be rewritten to use Megastore. In contrast, MongoTx provides MongoDB APIs with MongoTx API.

Google Spanner[3] provides distributed transactions based on two-phase commit and a SQL-like language to query data. A client of Spanner buffers dirty data and sends them when the client commits a transaction. Spanner is not public and the specification and implementation of the query are hidden. However, in theory, buffered dirty data at the client is not reflected in query results without sending them before each query or merging dirty data with query results returned by the spanner servers. Sending dirty data will have overhead for each query and merging dirty data requires query processing at the client.

WebSphere eXtreme Scale (WXS) [11] supports the both of the transactions and queries. WXS divides data to multiple partitions and provides single-partition and multi-partition transaction. The queries are written in a SQL-like language and WXS searches data in only one partition for each query. Therefore, to search data, clients must know all of the partitions in WXS.

Transactions on MongoDB have been introduced in some of open source communities[15][16][17]. In particular, as SD2V model, the data model with two versions in one document has been proposed[17]. In the data model, the safe version is stored in `value` field and the unsafe version is stored in `udpate` field. However, unlike SD2V, the data model is not compatible with the original MongoDB. In addition, the MongoDB queries are not supported in the implementation.

8. CONCLUSION

In this paper, we present MongoTx, which supports transactions and queries on the top of MongoDB. MongoTx uses MongoDB as its data store and supports MongoDB queries and MongoDB-like APIs with transactions. In addition, MongoTx supports most of the beneficial functions of MongoDB, such as indexing, sharding, replication, and the aggregation framework.

We evaluated the overhead of transactions and queries and the scalability of MongoTx with YCSB as a microbenchmak and TPC-C-like benchmarks as realistic workload. We confirmed, though the overhead was around 70% in the realistic benchmarks, the scalability was the same as MongoDB in any benchmarks.

9. REFERENCES

[1] Baker, J., Bond, C., Corbett, J. C., Furman, J., Khorlin, A., Larson, J., Léon, J.M., Li, Y., Lloyd, A., and Yushprakh, V. Megastore: Providing Scalable, Highly Available Storage for Interactive Services, In *Proceedings of the Conference on Innovative Data system Research* (CIDR 2011), Asilomar, CA, USA, 223-234.

[2] Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., and O'Neil, P. A Critique of ANSI SQL Isolation Levels, In *Proceedings of the 1995 ACM SIGMOD international Conference on Management of Data* (SIGMOD'95), ACM Press, New York, NY, USA, 1-10.

[3] Corbett, J. C. , et al., Spanner: Google's Globally-Distributed Database, In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation* (OSDI'12), USENIX Association (Oct. 2012), Hollywood, CA, USA, 251-264.

[4] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E., Bigtable: A Distributed Storage System for Structured Data, In *ACM Transactions on Computer Systems* (TOCS), vol.26 no.2, 2008, 1-26.

[5] Ferro, D. G., Junqueira, F., Kelly, I., Reed, B., and Yabandeh, M., Omid: Lock-free Transactional Support for Distributed Data Stores, In *Proceedings of IEEE 30th International Conference on Data Engineering* (ICDE 2014), IEEE (Mar. 2014), Chicago IL, USA, 2014, 676-687.

[6] Peng, D., and Dabek, F., Large-Scale Incremental Processing Using Distributed Transactions and Notifications, In *Proceedings of the 9th USENIX conference on Operating systems design and implementation* (OSDI'10), USENIX Association (Oct. 2010), Vancouver, BC, Canada, 1-15.

[7] Stonebraker, M. The Case for Shared Nothing. *Database Engineering Bulletin* vol. 9, no. 1, 1986, 4-9.

[8] Thusoo, A., Sarma, J. S., Jain, N., Shao, Z., Chakka, P., Anthony, S. , Liu, H., Wyckoff, P., and Murthy, R., Hive - A Warehousing Solution Over a Map-Reduce Framework, In *Proceedings of Very Large Data Bases* (VLDB 2009), vol. 2 no. 2 (Aug. 2009), 1626-1629.

[9] TPC benchmark C standard specification version 5.11, Transaction Processing Performance Council, 2010.

[10] IBM DB2. http://www.ibm.com/software/data/db2/

[11] IBM WebSphere eXtreme Scale. http://www-03.ibm.com/software/products/ja/websphere-extreme-scale/, 2005.

[12] MongoDB. http://www.mongodb.org/, 2009.

[13] Apache HBase. http://hbase.apache.org/, 2008.

[14] Apache Hive. https://hive.apache.org/, 2009.

[15] Perform Two Phase Commits, http://cookbook.mongodb.org/patterns/perform-two-phase-commits/

[16] Chess iX, Transaction in MongoDB, yes we can!, http://www.chess-ix.com/blog/transaction-in-mongodb-yes-we-can/.

[17] Optimistic transactions in MongoDB, https://github.com/rystsov/mongodb-transaction-example, 2013.