CONTENTS

Decongestant: A Breath of Fresh Air for MongoDB Through Freshness-aware Reads

Chenhao Huang
The University of Sydney
chenhao.huang@sydney.edu.au

Michael Cahill
MongoDB Inc
michael.cahill@mongodb.com

Alan Fekete
The University of Sydney
alan.fekete@sydney.edu.au

Uwe Röhm
The University of Sydney
uwe.roehm@sydney.edu.au

ABSTRACT

Many distributed databases deploy primary-copy asynchronous replication, and offer programmer control so reads can be directed to either the primary node (which is always up-to-date, but may be heavily loaded by all the writes) or the secondaries (which may be less loaded, but perhaps have stale data). One example is MongoDB, a popular document store that is both available as a cloud-hosted service and can be deployed on-premises. The state-of-practice is to express where the reads are routed directly in the code, at application development time, based on the programmers' imperfect expectations of what workload will be applied to the system and what hardware will be running the code. In this approach, the programmers' choice may perform badly under some workload patterns which could arise during run-time. Furthermore, it might not be able to utilize the given resources to their full potential – meaning database customers pay more money than needed.

In this paper, we present Decongestant: a system which will automatically and dynamically, as the application is running, choose where to direct reads, sending enough reads to secondaries when this will reduce load on a congested primary and boost the performance of the database as a whole, but without exceeding the maximum data staleness that the clients are willing to accept. A central insight is to use measured latency of read operations on primary and secondaries to determine whether the primary is congested.

In an experimental evaluation, we demonstrate our system adapts well to dynamically changing workloads, obtaining performance benefits when they can arise from use of the secondaries, while ensuring that returned values are fresh enough given client requirements. Our approach is decentralised and can be used by both cloud-consumers and on-premises users: it uses only client observations and the limited diagnostic data provided by the database to its clients.

1 INTRODUCTION

Distributed databases have become a popular offering due to their scalability and elasticity. Distributed databases typically deploy a combination of data sharding and replication over multiple nodes to provide both scalability and availability. Thus, there are typically multiple copies of data, and some distributed database offerings expose those to programmers via a performance tuning parameter where one can decide to which node (primary or secondary) the read requests should get routed. The challenge is to use this tuning knob wisely to balance the load, for good performance, while minimizing its pitfalls, namely access to potentially stale data. One typical example in this space is MongoDB, a popular NoSQL distributed database management system. MongoDB Atlas[1] is the corresponding cloud-hosted MongoDB service.

MongoDB internally is a classic log-based, primary-copy replication system. It is usually run as a replica set, where each node keeps a logical copy of the database [35]. Each replica set is a log-replicated state machine [34]. In cloud settings, all nodes are usually placed within one geographical region, but spread in different availability zones. There is one primary copy which processes all write operations. Each secondary copy pulls the updated log from the primary and then replays it to keep up with the primary. Thus, data on the secondary copies might be stale compared to that on the primary copy in a workload with frequent write operations. During a fail-over, one secondary copy is elected as the new primary.

Clients can direct read operations to either the primary copy or to a secondary copy (this is called *Read Preference* [24] and is indicated in the API as an optional parameter of a read request call). When reading from the primary copy on a healthy cluster with default settings, fresh data is returned. But if the primary copy is saturated, the read latency can be huge. In that case, a user can gain larger throughput and lower read latency by reading from secondary copies; however, data returned might be stale.

The state-of-the-art practice is to "hard-code" the Read Preference, making a choice explicitly when writing the application program. This is not ideal for several reasons. Firstly, developers may have insufficient information about workload and hardware capabilities for them to make a sensible choice when the code is written. Also, the "sensible" choice may differ over time as workload changes, but hard-coding is not able to adapt dynamically. Furthermore, the standard Read Preference options are limited to either primary or secondary, so that the nodes in a MongoDB cluster will never do the heavy-lifting together.

This paper shows how to capture knowledge *at run-time* of the current condition, and change the Read Preference dynamically, so that the overall performance of MongoDB can be improved. We adjust the proportion of secondary reads sent, in order to distribute work among the nodes of a MongoDB cluster. Our goal is to gain extra performance by reducing problems of congested nodes, so that the database can serve more clients without upgrading its hardware. This should help lower the cost of MongoDB for users. But we need to ensure each client sees data values that are "fresh enough" for the client's requirements, so we must avoid directing reads to a secondary when that node is too stale.

Achieving this is not easy. We must make the decision on where to direct a read on client-side information. Our approach

[1] https://www.mongodb.com/cloud/atlas

aims at both MongoDB cloud-consumers and on-premises users – there is no need to alter server code, or monitor the database's internal state. We only take measurements from the outside, as operations are submitted, or call the service API for the limited statistics it provides on the current status of servers. Our key insight to overcome this challenge is to measure the latency of read operations that were recently submitted by the client, for those sent to the primary and also for those sent to the secondary. In this paper we look at reads individually; if the client also needs session properties such as read-your-own-writes, or transaction-respecting snapshots, those can be obtained through capabilities of MongoDB, e.g. causal consistency[37].

Our key contributions are

- A mechanism to determine at run-time as either a cloud-consumer, or an on-premises user, whether the primary, or the secondaries, are currently congested with reads. This uses client-side measurements of the latency of recent read operations with each Read Preference, estimates the server-side component of that by subtracting network round-trip latency, and compares these estimates from the primary versus from the secondaries. This ratio is used in a simple feedback control. A high ratio indicates too much load on the primary, so more reads should go to the secondaries as long as those are not too stale. On a low ratio, indicating too much load on the secondaries, fewer reads should be sent to the secondaries in future.
- A system design and prototype implementation, called Decongestant, which uses this mechanism. It chooses at run-time to send a client's reads to primary or secondary as appropriate, for good performance while respecting a client-assigned limit on acceptable staleness of values returned.
- Experimental evaluation of Decongestant, to show that it maintains good performance while workloads vary (and it can do better than either hard-coded approach), and that it respects a client-assigned limit on staleness.

Our earlier poster paper [21] presented initial ideas in this direction. Here we adapt some of the text from there, describing motivation, background, and related work. Novel features of Decongestant, compared to [21], are: avoiding use of a secondary when it would give values that are too stale for client requirements (and never send reads to the secondaries when the clients can not tolerate any stale reads); capability to deliberately, in a controlled way, mix use of primary and secondaries in the same period, in order to outperform either hard-coded approach; improved latency estimate that separates network effects from those of server congestion; evaluations that show run-time adaptation to workloads that change in various aspects; a new S workload we use to demonstrate the validity of staleness reports from the servers (which Decongestant uses) as estimate of client-observed staleness.

The remainder of the paper is structured as follows. Section 2 introduces relevant concepts in MongoDB. Section 3 presents the design of Decongestant. We provide an evaluation of Decongestant in Section 4. Section 5 highlights, and contrasts with, related work. We conclude in Section 6.

2 BACKGROUND

Some text in this section already appeared in [21].

2.1 A brief overview of MongoDB and MongoDB-as-a-Service

As we have already discussed in section 1, MongoDB internally uses asynchronous primary-copy replication for fault-tolerance. This means it usually runs as a replica set. Fail-overs are rare [35]. This justifies our approach that exploits some of the capacity of the secondary nodes.

MongoDB also provides a sharding mechanism to enable horizontal scaling [22]. The entire database can be sharded into a number of shards, where each shard is a distinct subset of the whole database. Each shard can be deployed as a replica set, geographically far away from one another. MongoDB sharding is not used in this paper but the techniques we describe can be applied to sharded clusters, which support the same Read Preference API as standalone replica sets.

MongoDB Atlas is a "containerized" version of MongoDB, provided as a service. MongoDB Atlas has the same core as the open source Community Server, but with some additional, closed source functionality (e.g., security features). Customers are able to use MongoDB Atlas in a pay-as-you-go model. MongoDB Atlas is hosted on diverse cloud service platforms: Amazon Web Service (AWS), Microsoft Azure, or Google Cloud Platform (GCP). Customers can select the service locations, memory and storage size, number of vCPUs, etc. The database can be deployed with a few clicks. It is also trivial to scale-up if more computational power or storage are needed. MongoDB Atlas has various APIs for different programming languages and applications.

The main source of operational metrics offered by both MongoDB and MongoDB Atlas is via calling MongoDB `serverStatus` command. The MongoDB metrics can be queried frequently, but these deal only with internal properties, and do not report on hardware/OS/network aspects. MongoDB Atlas provides extra sources of operational metrics through MongoDB Atlas web API. This supplies some hardware metrics. However, those metrics only update once per minute, and can only be queried at a limited rate (100 requests per minute per project).

2.2 Read Preference

The Read Preference setting on read operations determines where they will be sent by MongoDB clients [24]. Read Preference options include `primary` (default), `primaryPreferred`, `secondary`, `secondaryPreferred`, and `nearest`. If the Read Preference `primary`, or `secondary`, is selected, then the reading request is directed to the primary copy, or to one of the secondary copies, respectively. Note that in MongoDB the users can not specify which of the secondary copies a read request will be sent to. `PrimaryPreferred` and `secondaryPreferred` provide users the option that in most cases the reads are sent to the primary or the secondary copies. However, in the situations where the preferred copy is not available, the reads are routed to the other option. When Read Preference `nearest` is chosen, the reading requests are directed to the nearest copy to the client based on client-measured network latency. The MongoDB client libraries periodically check which node is nearest. In our research, we use options `primary` and `secondary` to balance the workload among all MongoDB nodes.

When requesting a read on secondary copies, the clients can include a `maxStalenessSeconds` value to specify the maximum data staleness the client is happy to accept [25]. However, the `maxStalenessSeconds` value must be set to 90 seconds

or larger. As we will show later, Decongestant is able to bound the data staleness to substantially lower levels (eg 10 seconds).

The client driver randomly chooses a secondary copy to route a secondary read, as long as the latency between the secondary nodes do not differ by more than 15 milliseconds [28]. In our experiments, all secondary reads are directed to one secondary copy randomly.

There is another tuning knob in MongoDB called Read Concern which determines the durability, consistency, and isolation properties of the data read from MongoDB [23]. We use `local` Read Concern, which is the default setting, in all our experiments.

2.3 Data staleness in MongoDB

Let's first look at how MongoDB performs a write operation. When a write request reaches the primary copy of MongoDB, there will be an atomic transaction issued doing two things: 1) applying the database operation to the primary; 2) recording operations on the primary's operation log (called `oplog`). Once that transaction commits on the primary (and after waiting for any other transactions with earlier `oplog` entries to also commit), the `oplog` entry is visible to secondaries, and it will then be pulled by the secondaries and written to their `oplog`. After that the secondaries will apply the operations. Each node records the timestamp of the latest `oplog` applied (called `lastAppliedOpTime`). The `lastAppliedOpTime` of each node are known by all others.

By calculating the difference between the `lastAppliedOpTime` of a secondary node and the `lastAppliedOpTime` of the primary node, we can estimate the data staleness of the secondary node. Since the `lastAppliedOpTime` of one node is known by all other nodes, the comparison can take place at any of the MongoDB copies. This information can be provided to a client in the `serverStatus` command.

There is a potential source of error here. The `lastAppliedOpTime` for a secondary copy as recorded on the primary copy, might be earlier than the truth. This is because that the latest `lastAppliedOpTime` of the secondary copies might not yet have been transmitted to the primary copy. So the data staleness of the secondary, when calculated using statistics reported by the primary copy, might be larger than reality. Similarly, the `lastAppliedOpTime` of the primary copy as known on some secondary copy may be earlier than the current truth. So data staleness as calculated from the status at a secondary node can be smaller than reality. In Decongestant, we use the `serverStatus` information from the primary for this calculation, to be conservative in enforcing a client-requested limit on staleness.

3 DESIGN OF DECONGESTANT

In this section, we describe the design of Decongestant, an automated system to direct the reads dynamically for MongoDB during run-time of the application.

Our design needs to overcome some challenges. The foremost challenge is to have a single mechanism that can detect a variety of congestion situations. The bottleneck resource of a node of MongoDB, when it saturates, varies with different DBMS's hardware, configurations, and workload. It may be CPU usage, memory usage, even having a large number of Write Ahead Logs (WAL) waiting to be flushed to the disk. We also need a mechanism to detect when a secondary is too stale to be usable, considering the client's freshness requirement. An important challenge is that our mechanisms need to use only information

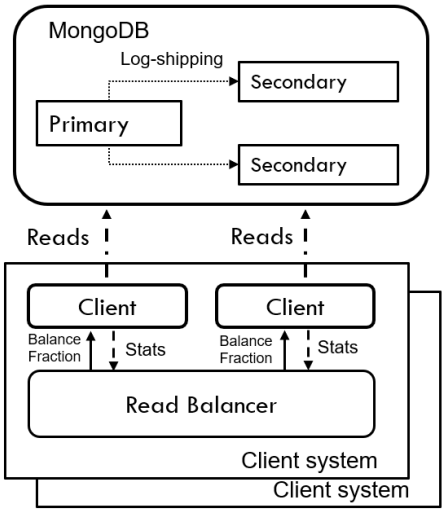

Figure 1: A simplified architecture of Decongestant

readily available to the consumers of the MongoDB, as we do not do any modification of the server. On the client-side, we can observe requests and responses for the operations submitted to the service, and we also have some limited access to server information through available status reports (but this is much less detailed than what a server-side design could exploit).

In this section, we first present an overview of Decongestant and we briefly introduce the decision-making component of the system, called Read Balancer. Then, we describe how the client application works in Decongestant. After that, we depict how Read Balancer does its job. Finally, we discuss some of the design details and considerations of Decongestant.

3.1 System Architecture

Figure 1 shows a simplified architecture of Decongestant. A component called Read Balancer resides on each client system where (maybe multiple) client applications are executing. The Read Balancer decides the percentage of read operations to be sent to the secondary copies. This percentage is to be chosen so as to improve the overall performance, by redirecting reads when one of the servers is congested, but not using secondaries at all if their data would be too stale to meet the client's specified demand for data freshness. The Read Balancer communicates with the clients via a few shared variables:

- The latest decision for each client on what percentage of the read-only transaction should be directed to the secondary copies, called *Balance Fraction*. The range of the Balance Fraction is between 10% to 90%, inclusive; or 0%. Balance Fraction is set to 0 when Decongestant estimates some secondary's data staleness exceeds the client's limit, so they will stop sending any read requests to the secondaries till this is remedied. The clients can express that they are not willing to accept any stale data, by setting the data staleness limit to 0.
- Two lists which keep track of client-observed transaction latencies (for reads that were sent to Primary and to Secondaries, respectively).

3.2 How do the client applications work in Decongestant?

In Decongestant, a client is expected to cooperate with the Read Balancer, as follows. Before invoking any read-only transaction call, the client first examines the most recent decision on the Balance Fraction from the Read Balancer. Then the client should flip a biased coin, and with a probability equal to the Balance Fraction, the call should to be directed the secondary copies of MongoDB. The clients keep track of the latency of their read-only transactions, and report them to the Read Balancer through one of the shared queues (depending on which Read Preference was actually used in the call).

3.3 How does the Read Balancer work?

In this section, we show how the Read Balancer acts. We first provide a high level overview of the Read Balancer and how it changes the Balance Fraction. Then, we discuss some technical details and considerations. Algorithm 1 shows the pseudo code for Read Balancer.

The Read Balancer periodically updates suggestions on the Balance Fraction, i.e. the percentage of read-only transactions that should go to the secondary copies. On system start up, the Read Balancer initializes the Balance Fraction as 10%. The Balance Fraction will be updated periodically (every 10 seconds in our implementation). We keep the Balance Fraction so it is never 0, except when this is needed because a secondary is too stale to use given the client-defined limit, or when the clients are not willing to accept any stale data. Similarly, Balance Fraction is never 100%. The reason to exclude the extreme values is to ensure that some reads are regularly using each server, so that the system has up-to-date data on the situation of the servers.

Within each period (when there is no need for the Read Balancer to recalculate its recommended fraction), the Read Balancer pings all nodes of the MongoDB cluster regularly, in order to record the Round Trip Time (RTT) between the client system and each node. In addition, the Read Balancer queries the primary node of the MongoDB replica set once per second, using the `serverStatus` command to get the latest data staleness estimation of each secondary node. Whenever the data staleness estimation on *any* secondary node exceeds the maximum data staleness the clients are willing to except, the recommended Balance Fraction will become 0, and so all read requests will be directed to the primary, until the data staleness situation on secondary nodes improves. (See detailed discussions in section 4.)

By the end of each period, the Read Balancer retrieves, from the shared lists, the recorded latencies of those read-only transactions that were sent to the primary during the period, and it calculates the median ("P50") value of these latencies; similarly it takes the list of latencies for read-only transactions sent to the secondaries, and determines the median of those. The Read Balancer then calculates a value called Server-Side Latency, for the reads on primary and secondaries, respectively. Server-Side Latency for a corresponding Read Preference option equals the median ("P50") latency of the read requests with that Read Preference option minus the median ("P50") Round Trip Time (RTT) of all MongoDB nodes corresponding to that the Read Preference option. See subsubsection 3.3.1 for details. These numbers act as estimates for the delay experienced by operations as they are performed on the server, including time spent within executing the MongoDB instance and also delays in the operating system or disk. The intuition here is that when a node is congested, this

Algorithm 1: Algorithm for Read Balancer

SharedVars: StaleBound - Client-set limit on data staleness
$L_{client,primary}$ - list of client-observed latencies of primary-sent ops
$L_{client,secondary}$ - list of client-observed latencies of secondary-sent ops
Bal - current Balance Fraction; initially LOWBAL

PrivateVars: RecentBal - list of 4 recent periods' Balance Fractions; initially all LOWBAL
Staleness - array of staleness reported by primary for each secondary
RTT - array of round-trip times to each server

Constants: DELTA - one-period change in Balance Fraction (10%)
LOWBAL - lowest value for non-zero Balance Fraction (10%)
HIGHBAL - highest value for Balance Fraction (90%)
LOWRATIO - latency ratio above which we increase Balance Fraction (0.75)
HIGHRATIO - latency ratio below which we decrease Balance Fraction (1.3)

```
1  Function Rcv-ServerStatus():
2      Update Staleness from ServerStatus
3      if (StaleBound == 0) or (max(Staleness) > StaleBound) then
4          Bal ← 0
5      else
6          Bal ← RecentBal.latest()
7      end
8  end
9  Function OnPeriodEnd():
10     L_ss,primary ← P50(L_client,primary) - P50(RTTprimary)
11     L_ss,secondary ← P50(L_client,secondary) - P50(RTTsecondary)
12     Ratio ← L_ss,primary / L_ss,secondary
13     if Ratio > HIGHRATIO then
14         NewBal ← min(RecentBal.latest() + DELTA, HIGHBAL)
15     else if Ratio < LOWRATIO then
16         NewBal ← max(RecentBal.latest() - DELTA, LOWBAL)
17     else if All RecentBal entries are the same then
18         NewBal ← max(RecentBal.latest() - DELTA, LOWBAL)
19     else
20         NewBal ← RecentBal.latest()
21     end
22     RecentBal.dequeue().enqueue(NewBal)
23     if (StaleBound == 0) or (max(Staleness) > StaleBound) then
24         Bal ← 0
25     else
26         Bal ← RecentBal.latest()
27     end
28 end
```

figure will increase, no matter whether the bottleneck resource is in MongoDB or elsewhere on the server.

The Read Balancer uses the ratio of the Server-Side Latency of reads on the primary, to the Server-Side Latency of reads on the secondary. To achieve the maximum performance possible of the MongoDB Cluster, all the nodes in the cluster should do the heavy-lifting together, sharing the work. This means, ideally, the ratio should be close to 1. If the ratio is much larger than 1, it means the primary copy is congested compared to the secondary copies. In this case, the Read Balancer increases the Balance Fraction for the next period, directing more reads to the secondary nodes. On the other hand, if the ratio of Server-Side Latency is much less than 1, indicating that the secondary nodes are congested more than the primary nodes, then the Read Balancer would decrease the Balance Fraction, sending less reads to the secondary copies in the next period. If the ratio is close to 1 and it has been close to 1 for quite a while, Read Balancer would also decrease the Balance Fraction to explore "downward". This is to make sure the reads go to the primary node as much as possible, in order to improve the data freshness and avoid potential stale reads [26]. Unlike our previous work [21], the Read Balancer does not look for one "correct" Read Preference in each period. Instead, it tries to balance the workload among all the nodes to achieve the best performance possible.

3.3.1 Server-Side Latency. As discussed before, the Read Balancer utilises Server-Side Latency to make decisions. Server-Side Latency of a Read Preference option is found as follows: we take the median latency of all read-only invocations with that Read Preference as observed on the client side in the previous period, minus the median Round Trip Time (RTT) of all MongoDB nodes corresponding to that Read Preference choice.

$$L_{ss} = P_{50}(L_{client}) - P_{50}(RTT)$$

In MongoDB, all nodes are spread across different availability zones within a region as much as possible, when deployed in a public cloud. The Round Trip Time (RTT) between a certain client and nodes in different availability zone varies. Although the difference is usually less than 2 milliseconds, it is enough to impact the latency observed on the client side for those workloads with light read-only transactions, such as YCSB, where the latencies themselves are sometimes only 1 to 2 milliseconds.

3.3.2 Bounded Data Staleness. The Read Balancer also talks to the primary node of the MongoDB replica set several times each period, to call `serverStatus` at the primary. It uses this information as described in subsection 2.3 to conservatively estimate the data staleness of each secondary node.

The clients can tell the Read Balancer the maximum data staleness they are happy to accept. As we have explained before, the MongoDB clients can only choose to read from either the primary node or from any secondary node, they can not specify which secondary node they would like the read requests to be sent to. So, as long as the Read Balancer finds the estimated data staleness of any secondary node is larger than the threshold set by the clients, the Read Balancer immediately notifies every client, that all future read requests should only be sent to the primary copy - the Balance Fraction is set to be zero. The Read Balancer resumes a non-zero Balance Fraction once the maximum estimated data staleness of the secondary copies drops below the threshold set by the clients.

One concern may be raised here. The Read Balancer restarts sending read requests to the secondary nodes once the maximum data staleness of all secondary nodes drops below the threshold. Would that extra work cause the data staleness to quickly go beyond the limit again? We had the same concern. Our empirical study shows that in MongoDB, data staleness increases gradually on the secondaries, but when it goes down, it drops swiftly, to nearly zero. The high-level idea is that the gradually increasing data staleness is caused by a congested primary node, which is too busy processing data requests so it is not able to provide the `oplog` to the secondary copies. But once the `oplog` is sent, the secondary nodes catch up quickly. See Section 4.5.

The data staleness for secondaries which is estimated by this method may be larger than what a client actually observes. As well as the possible overestimate from the primary copy not yet knowing very recent activity updating the secondary, there is also the possibility that there are `oplog` entries not yet applied to the secondary copies but these might not modify the particular data the MongoDB client queries. We are conservative, and avoid using a secondary if our (perhaps over-) estimate breaches the client-set limit on staleness. Section 4.5 reports experiments to check the alignment between the data staleness estimate used in our decision, and measurements in a targeted S-workload that observes latency at clients.

4 EVALUATION

In the following, we present an evaluation of Decongestant. We first introduce the methodology and settings we use in Section 4.1. We then demonstrate Decongestant's ability to detect and adapt to variation of workloads in Section 4.2. We explore Decongestant's ability of balancing the load among all MongoDB nodes to achieve better performance than using current practice in a read-intensive workload (Section 4.3), and Decongestant's capability of trading data freshness for performance in the workloads with a mixture of reads and writes (Section 4.4). Section 4.5 covers Decongestant's competence of bounding the data staleness. Finally, we show that running S workload concurrently has low impact on performance measurements of standard workloads, in Section 4.6.

4.1 Method

4.1.1 Platform. The experiments are executed on AWS. The MongoDB clients are on an AWS c4.4xlarge instance (16 vCPUs and 30 GB RAM), located in the region ap-southeast-2.

We deploy our own MongoDB cluster on AWS, in order to maintain consistent results independent of infrastructure and software version changes that are out of our control in MongoDB Atlas. We replicate the configurations from MongoDB Atlas in June 2020. The MongoDB version is 4.2.6. A 3-node MongoDB cluster is deployed on 3 AWS r4.2xlarge instances, which has 8 vCPUs and 61 GB RAM, located in the same region as the clients but different Availability Zones: ap-southeast-2a, ap-southeast-2b, and ap-southeast-2c, respectively.

4.1.2 Decongestant settings. In all following experiments, Decongestant considers the ratio of Server-Side Latency of the reads on the primary to the Server-Side Latency of the reads on the secondaries as being normal when the value ranges from 0.75 to 1.30. When the ratio is greater than 1.30, Decongestant considers that the primary is congested, and thus it increases the percentage of reads sent to the secondaries in the next period by 10%. A ratio less than 0.75 leads Decongestant to send 10% fewer reads to the secondary nodes in the next period, as it shows that the secondaries are more congested. Balance Fraction starts at 10%

Table 1: Percentage of transactions in the original TPC-C workload versus the read-write TPC-C workload used in the experiments.

	TPC-C	Read-Write TPC-C
Stock Level	4%	**50%**
Delivery	4%	4%
Order Status	4%	4%
Payment	43%	**20%**
New Order	45%	**22%**

and is capped at 90%. Decongestant revisits the Balance Fraction decision every 10 seconds. In our experiments, unless stated explicitly, the maximum data staleness the clients are willing to accept is set to be 10 seconds.

Read Balancer keeps the four previous records of the Balance Fraction. If they remains the same, Read Balancer pushes down the Balance Fraction by 10% in the next round.

4.1.3 Baselines. As well as showing the performance of our prototype system Decongestant, we also look at two baselines corresponding to current practice. In these, the clients run against MongoDB, without any Read Balancer installed, or any of the possible overheads of communicating with it; in the baseline Primary, each read is hard-coded with Read Preference set to `primary`. In baseline Secondary, each read is hard-coded with Read Preference as `secondary`.

4.1.4 Workloads. Two different sets of transactions are used here to evaluate the performance of Decongestant. We use YCSB [12] with varying read-write percentage: YCSB-A (50% reads and 50% writes) and YCSB-B (95% reads and 5% writes). YCSB represents a light-weight workload, with simple get and put operations. As a more demanding workload, we also use a variant of TPC-C (we call it read-write TPC-C). This uses the transactions from TPC-C, but unlike write-heavy traditional TPC-C, we aim here for a balance between read-only and update transactions. To do so, the percentage of Stock Level transaction, which represents a read-only transaction, is set to 50%. Table 1 shows the detailed breakdown of the read-write TPC-C workload as compared to standard TPC-C. The TPC-C queries are implemented by Kamsky [29] who has adapted the TPC-C benchmark to the MongoDB query language and transaction semantics, as well as adapting it to MongoDB's best practices.

4.1.5 S workload to monitor data staleness. In order to check whether Decongestant keeps the maximum data staleness promise, we need a method to sample the data staleness from the client side. While Decongestant uses estimates for staleness based on MongoDB's internal information on the status of oplog application, we want to validate our system's success using real measurements. So, we propose S workload (S stands for staleness). The S workload could be run standalone, but, in all our experiments[2], we generate this S workload alongside the main performance-focused OLTP workloads, such as YCSB, TPC-C.

The high-level idea of S workload is similar to our previous work [20, 40]. The S workload includes two workers (each worker can be implemented as a separate process or a thread): one writer and one reader. The job of the writer is to keep writing the current timestamp to a dedicated item in the database at a high frequency. It is not necessarily to write as fast as possible, but it should work at least as fast as the reader does.

Periodically, the reader probes the contents from the various copies of the dedicated cell. In each probe, the reader sends out two read requests: one with the Read Preference Primary and the other with the Read Preference Secondary. The reader records the results. Then, in the analysis phase (after the experiment), we can determine the freshness of the data returned from the secondary reads by comparing the results between the primary read and the secondary read. A slight variation is done at times when the main application is not using the secondaries at all (and so clients will see no staleness at these times): the second read in each probe of S workload can be simply directed to the primary copy again.

The staleness monitoring approach used by S workload is a bit different from the one used by [20, 40]. In those prior works, the reader only sends one read request in each probe, with the Read Preference Secondary; during the analysis phase, the timestamp when the read is sent and the read value are compared, to determine the staleness gap. The assumption in their approach was that the timestamp in the dedicated cell in the object database keeps advancing smoothly. This assumption doesn't always hold when monitoring is run together with another existing OLTP workload. There are times when a write takes a long time to finish, causing the value in the dedicated cell of the primary copy to be unchanged for a while. By definition, the value in the primary copy is fresh. However, the timestamp when the read request is sent keeps going forward as the S-reads are generated continuously. So, if the approach described in [20, 40] is used, fake staleness might be reported.

Note that the data staleness measured by this S workload might be larger than the data staleness seen by any given application client, as S workload frequently updates the item being read, while some application read may be on slowly-changing data, where the returned value is correct even if the secondary is far behind in applying the oplog. Still, if we succeed in bounding staleness seen in S reads, we can be sure that any other application also has data that is at least this fresh.

4.1.6 Measurements. The experiments have two distinct styles. In some, we report on the time-varying properties through a run of a system when a workload is applied (perhaps the workload changes at specific points during the run). In those experiments, we report for each separate 10 second period on throughput of appropriate transactions during the period, P_{80} latency (that is, the time within which were completed 80% of reads of the period), the percentage of reads that were sent with Read Preference Secondary during the period, and sometimes data staleness, either as measured by S workload, or as estimated conservatively using reports from the primary of the max difference between any secondary's `lastAppliedOpTime` and that of the primary. Although Server-Side Latency is used by Read Balancer to make decisions, all latency reported in the following figures are end-to-end latency observed by the clients. Measuring the actual percent of reads sent to the secondary copies is done by counting the read operations which were sent to the primary copy and those sent to secondary nodes; we do not simply echo what Decongestant suggests through Balance Fraction.

Other experiments give a single data point for overall performance (throughput, latency or staleness) for runs in some situation, typically shown in a figure where some important parameter of the workload (eg number of clients) varies along the

[2] except in one experiment of Section 4.6.

x-axis. In these experiments, except where stated explicitly, each data point is taken from the average over 3 runs; in each run, measurements exclude the first 100 seconds, which is treated as a warm-up period.

4.2 Adapting to dynamic workload

We first claim that Decongestant is able to adapt well to variation of workloads. We compare it to the outcomes for the two baseline systems, where all reads have hard-coded read preference setting, so either all go to the primary, or all to the secondaries.

Figure 2 shows the throughput of read operations, 80-percentile latency (end-to-end) of read operations, and the actual percentage of reads which are sent to the secondary copies, during a run with a dynamically-changing workload. The workload in this run starts with YCSB-A (50% reads) with 180 clients, and swaps to YCSB-B (95% reads) at the 620^{th} second. S workload is running as well, throughout. For the first 90 seconds, Decongestant warms up shifting from its initial setting with 10% of reads on secondaries, over time sending more and more read operations to the secondaries (Figure 2 (c)), until the highest amount we allow (90%) of the reads are secondary ones. During this period, the throughput increases (Figure 2 (a)) and the 80-percentile latency drops (Figure 2 (b)). From the 90^{th} second to the 620^{th} second, the percentage of reads sent to the secondary copies stabilises at 90%. This is the same performance we achieved in previous work [21]. At the 620^{th} second, the workload shifts from YCSB-A (50% reads) to a read-dominated YCSB-B (95% reads). Decongestant quickly responds by sending less reads to the secondary. The percentage of reads routed to the secondary nodes stables at 70% under this workload. The intuition is that the primary node deals with 5% writes and a bit less than 1/3 of reads, and two secondary nodes process between them a bit more than 2/3 of the reads. Recall that our MongoDB cluster is a three-node cluster with the same capacity in each node. This shows that Decongestant successfully balances the read load proportionally to the capacity behind each Read Preference option. As a result, the throughput (Figure 2 (a)) of the read operations is higher than either baseline (only sending read requests to the primary or secondaries); and the 80-percentile (Figure 2 (b)) latency is better than baselines.

Once the system has adjusted to the changed workload, the ratio of Server-Side Latency of the reads on the primary to the Server-Side Latency of the reads on the secondaries remains between 0.75 to 1.30. During this period, Read Balancer tries to push more reads to the primary on every fifth period. But as the Read Balancer finds it does not work well, the Read Balancer bring the Balance Fraction back to 70%.

Figure 3 is another example showing that Decongestant successfully notices and adjusts to the workload shifts; here both read-intensity and total load change. At the beginning the workload is YCSB-B (95% reads) with 180 clients. Then, after 230 seconds, it shifts to YCSB-A with 20 clients. On system start up, Decongestant increases the percentage of reads sent to the secondaries (Figure 3 (c)). The percentage soon reaches an optimised state at 70%. During this period, the throughput of read operations in Decongestant is higher (Figure 3 (a)) and the 80-percentile latency of those reads (Figure 3 (b)) is lower than when the Read Preference is hard-coded as Primary or Secondary. At the 230^{th} second, the workload switches to YCSB-A (50% reads) with 20 clients. Decongestant quickly decreases the percentage of reads sent to the secondary, as the primary can now handle all the load. The allocation becomes stable at the minimum we allow (10%), to make sure Read Balancer keeps getting enough recent information on the state of the secondaries, so we will detect future congestion if it were to happen.

Figure 4 shows the performance of Decongestant with the dynamic read-write TPC-C workload. Unlike previous experiments on YCSB, the throughput and 80-percentile latency (end-to-end) here are reported for each 1 minute interval, and the actual percentage of secondary Stock Level transactions are recorded every 10 seconds. The workload starts with 20 clients, and then at the 5^{th} minutes the client number increases to 200. After 5 more minutes, it goes down to 20 clients. Figure 4 (c) shows the actual percentage of secondary Stock Level transactions. It starts at around 10%. From the 5^{th} minute, Read Balancer is able to notice the high contention environment, and quickly pushes up the percentage of secondary Stock Level transactions. This soon brings the throughput (Figure 4 (a)) and 80-percentile latency (Figure 4 (b)) of Decongestant to a level similar to the situation where the Read Preference is "hard-coded" as Secondary. During the 5^{th} minute to the 10^{th} minute, there are a few downward spikes in the actual percentage of secondary Stock Level transactions. They are caused when the maximum data staleness on the secondary copies exceeds the clients' threshold, which is 10 seconds; this is detected by Read Balancer, so we stop sending Stock Level transactions to the Secondary copies. The measured percentage is not 0, as the staleness check is run once per second, while the percentage reported here is taken over 10 seconds. The pink vertical lines in the figure shows the seconds where all reads are directed to the primary, due to exceeding data staleness. After the 10^{th} minute, the number of clients drops to 20. Read Balancer gradually brings back most of the Stock Level transactions to the now-uncongested primary node, to provide lower staleness.

4.3 Achieving better performance by sharing load in read intensive workloads

In this section, we show how Decongestant's capability of balancing the read load among *all* the nodes of MongoDB cluster, achieves a better peak performance for read-intensive workloads, compared to hard-coding Read Preference as either Primary or Secondary. Each hard-code approach leaves some node underloaded. Decongestant does not try to specially identify whether the workload is read intensive; instead, the exact same approach is used throughout. Each data point in a plot is the average over three runs, excluding the warm-up period.

Figure 5 shows the throughput of reads, 80-percentile latency (end-to-end) of reads, and the actual percentage of reads sent to the secondary copies, plotted against number of clients, in YCSB-B (95% reads). We first discuss the actual percentage of reads sent to the secondary copies (Figure 5 (c)) over a varying number of clients. We can see with a low load of between 10 to 50 clients, Decongestant sends most read requests to the primary node. Decongestant sends more reads to the secondary copies, with the percentage growing corresponding to client number, in the range between 50 to 100 clients. The percentage of secondary reads is roughly stable at around 70% when the number of clients ranges from 120 to 200. This means, in a MongoDB cluster with 3 nodes of the same capacity, the primary nodes deals with 5% write operations and around 30% read operations; while the two secondary nodes between them process 70% read requests. This shows that all three nodes do the heavy-lifting together.

When the client number is between 120 to 200, all three nodes work together, instead of the primary node or the secondary

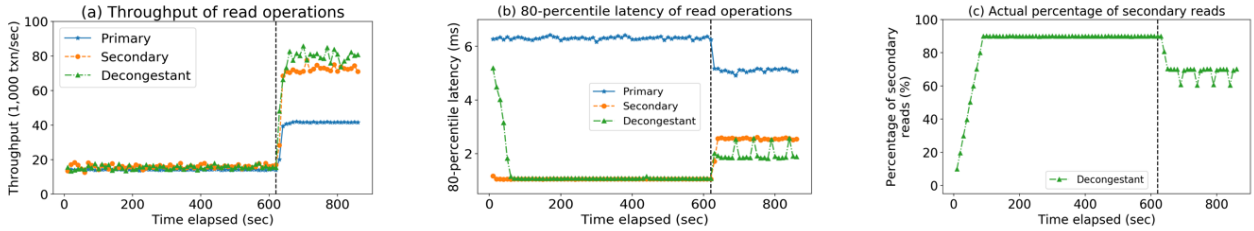

Figure 2: Decongestant's ability to respond to sudden increase of the read-write ratio in YCSB. The plots show the throughput of read operations, 80-percentile latency (end-to-end) of read operations, and the actual percentage of reads sent to the secondary copies, in a dynamic workload. The dynamic workload starts with YCSB-A (50% reads) with 180 clients, and swaps to YCSB-B (95% reads) at the 620th second. The vertical dotted line shows the time when the variation happens.

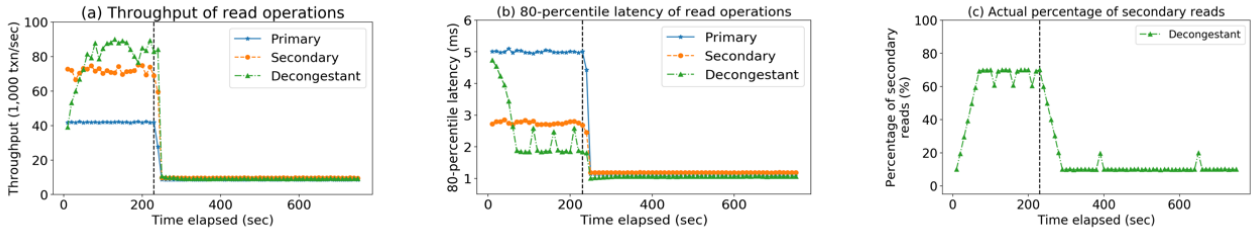

Figure 3: Decongestant's ability to respond to sudden decrease of both the read-write ratio and the number of clients, at the same time, in YCSB. The plots show the throughput of reads, 80-percentile latency (end-to-end) of reads, and the actual percentage of reads sent to the secondary copies, in a dynamic workload. The dynamic workload starts with YCSB-B (95% reads) with 180 clients, and swaps to YCSB-A (50% reads) with 20 clients at the 230th second (indicated by the vertical dotted line).

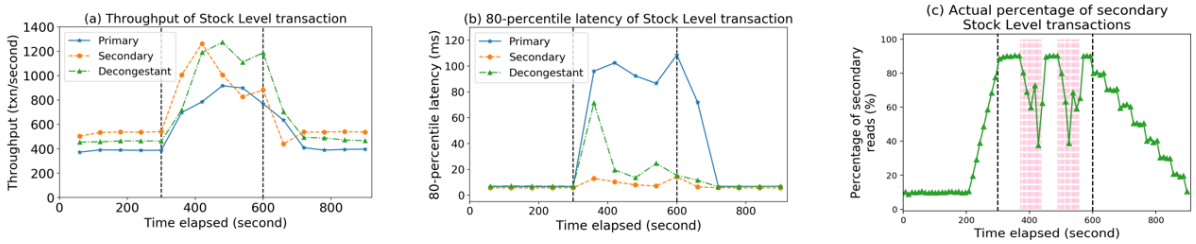

Figure 4: Decongestant's response to variations of client number in read-write TPC-C. The client number starts at 20. It bursts to 200 clients at the 5th minute. The client number drops back to 20 at the 10th minute. The black vertical dotted lines show the times when the variations of the client number happen. The plots display the throughput of Stock Level transactions, their 80-percentile latency (end-to-end), and the actual percentage of Stock Level transactions sent to the secondary copies. The throughput and the 80-percentile latency (end-to-end) are reported on per-minute bases, while the actual percentage of secondary Stock Level transactions are recorded once per 10 seconds. The pink vertical dotted line shows the seconds where all reads are directed to the primary, due to exceeding data staleness.

nodes only, so it is not surprising that Decongestant is able to achieve a throughput (Figure 5 (a)) that is around 30% higher than only routing the read requests to the secondary copies, and around 2.5 times than only sending read operations to the primary node. We also see that the 80-percentile latency for Decongestant is lower that the two hard-coded systems when the load is high.

We do not plot data staleness for YCSB-B (95% reads) here. The 80-percentile data staleness, measured both by S workload and Decongestant, for YCSB-B are constantly zero in all our experiments. This situation is not very surprising, as there are only 5% of writes in the workload. There are occasionally one or two data staleness values observed to be one second by the S workload, when the Read Preference is "hard-coded" as Secondary. Since the granularity for the data staleness is one second, we do not feel it is very meaningful.

4.4 Trading data freshness for performance

In this section, we show that Decongestant is able to trade data freshness for performance when needed in the workloads with a mixture of reads and writes. Again, we point out that Decongestant never tries to identify whether a workload has a mixture of

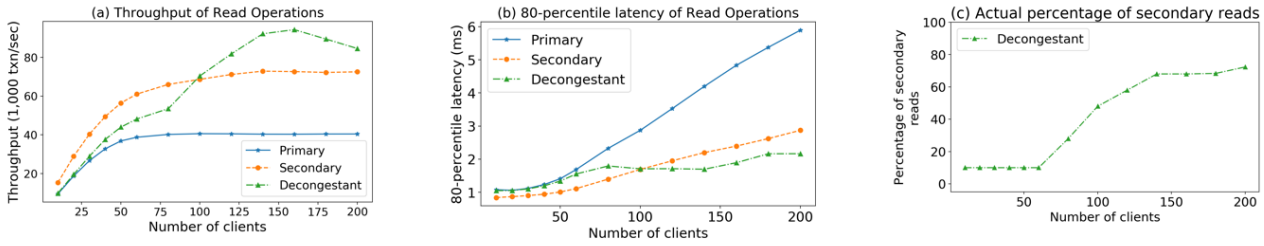

Figure 5: Performance trends with increasing client count. The plots show the throughput of reads, their 80-percentile latency (end-to-end), and actual percentage of reads sent to the secondary copies, against the number of clients in YCSB-B (95% reads).

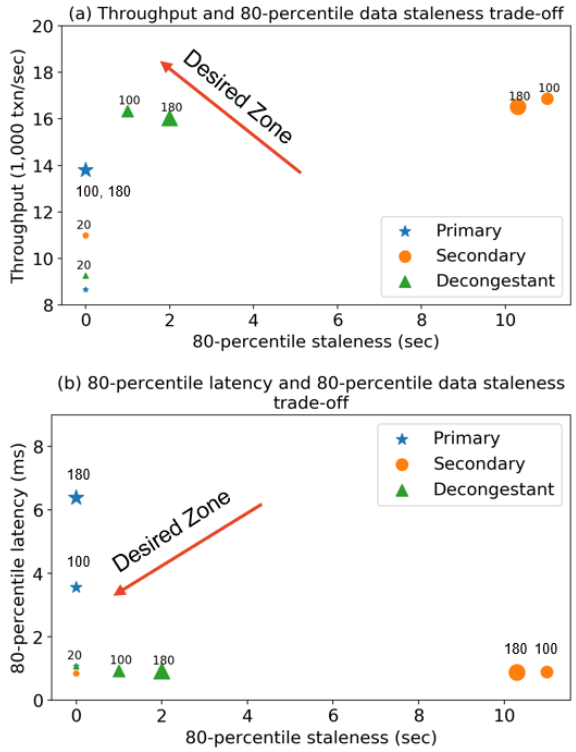

Figure 6: Performance and 80-percentile data staleness trade-off in YCSB-A for read operations. (a) shows the trade-off between the throughput and the 80-percentile data staleness of the read requests. The number above the marks and the size of them indicates the number of clients. The upper left corner is the desired area for Decongestant, with large throughput and small data staleness. (b) shows the trade-off between 80-percentile latency (end-to-end) and 80-percentile data staleness for read operations. The lower left corner is the target zone for Decongestant, with small latency and data staleness.

reads and writes or it is a read-intensive one: the same method works for either scenario. The experiments in this section are each run three times for each setting, and the average value is taken and shown as a point in the charts.

Figure 6 shows the performance and the data staleness trade-off for YCSB-A (50% reads). To avoid the figures being too cluttered, we only include 3 groups of data points here. The number above the marks and the size of them indicates the number of clients. Three client numbers are chosen: 20, 100, and 180, representing light, medium, and heavy load, respectively. Figure 6 (a) demonstrates the trade-off between the throughput of reads and the 80-percentile data staleness of them. The upper left corner is the desired zone in Figure 6 (a), with small data staleness and large throughput. When the load is low (20 clients), the data points for the three situations, whether always directed to either the primary or secondary nodes or with Decongestant, are close. When the load is medium (100 clients) and large (180 clients), Decongestant is able to push the data point close to the desired zone while the baselines are far away (Primary having low throughput, and Secondary seeing high staleness).

Similar conclusions can be reached for the trade-off between latency and data staleness from Figure 6 (b). The lower left corner is now the ideal area, as it shows small values in both latency and data staleness. The advantage of using Decongestant is clear, offering a sweet spot in terms of 80-percentile latency and 80-percentile staleness.

Figure 7 shows the performance and the data staleness trade-off in a read-write TPC-C workload (50% reads). Figure 7 (a) reports the trade-off between the throughput of the Stock Level transaction and the data staleness, where the upper left corner is desirable. Figure 7 (b) depicts the trade-off between latency of Stock Level transactions and data staleness, where the lower left corner is the target. Decongestant is able to push the data points toward the desired zone.

4.5 Bounding data staleness

In this section, we discuss the data staleness issue. We make two claims here:

- Decongestant's estimation of data staleness, from `serverStatus` reports, closely aligns with the data staleness seen by the clients.
- When the maximum data staleness of a secondaries copy exceeds the threshold set by the clients, the clients of Decongestant will not see this.

The largest data staleness the clients are willing to accept is set to 10 seconds, and we measure the staleness seen by clients, with S workload from Section 4.1.5.

Figure 8 compares the data staleness from `serverStatus` to the one seen by the clients, against time elapsed. The workload is one run of YCSB-A together with S workload, with. 100 clients. This figure shows that maximal data staleness known by the Decongestant via MongoDB `serverStatus` (and used by Decongestant to detect cases where excessive staleness means that reads should avoid the secondary nodes), aligns quite well

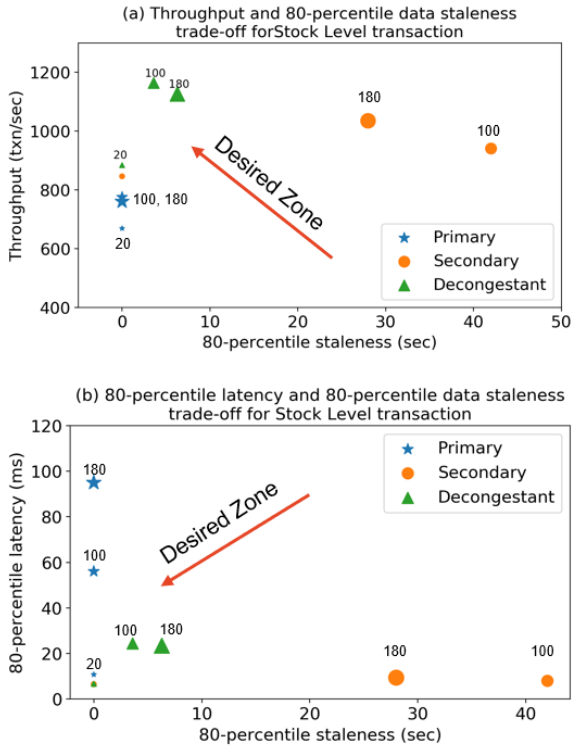

Figure 7: Performance and 80-percentile data staleness trade-off for Stock Level transacts with read-write TPC-C. (a) shows the trade-off between throughput of Stock Level transacts and 80-percentile data staleness. The number above the marks and the size indicates the number of clients. The upper left corner is the desired area for Decongestant with both large throughput and small data staleness. (b) shows the trade-off between 80-percentile latency (end-to-end) of Stock Level transactions, and 80-percentile data staleness. The lower left corner is the target zone for Decongestant with both small latency and data staleness.

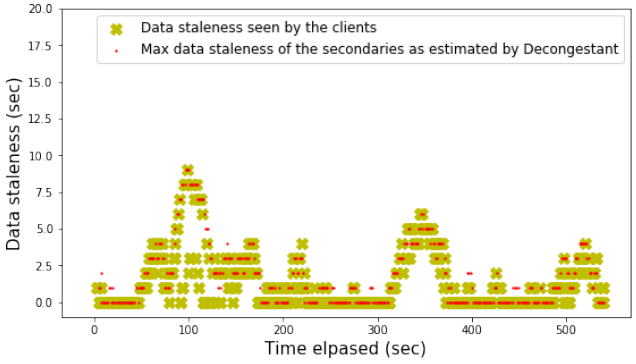

Figure 8: The maximum data staleness of the secondaries estimated by the Decongestant versus the one seen by the clients, against time elapsed. The workload is YCSB-A together run with S workload, from 100 clients. The data staleness estimated by Decongestant is good as long as it is not smaller than the one seen by the clients.

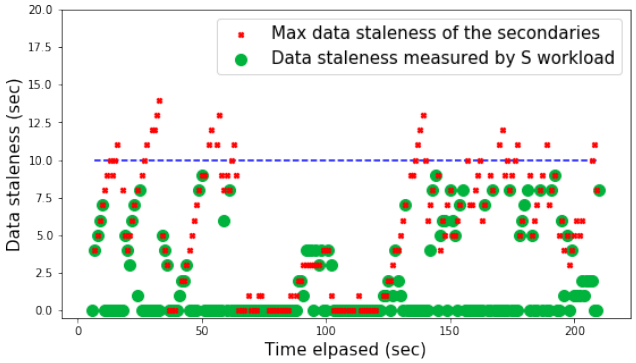

Figure 9: Data staleness measured by S workload versus the max data staleness of the secondaries in Decongestant, with the data staleness limit set to be 10 seconds. The workload used here is read-write TPC-C with 60 clients.

with the data staleness seen by the clients. In some cases, we can see that the data staleness estimated by the Decongestant is larger than the ones seen by the clients. This is acceptable, as the Decongestant records the maximum data staleness among all secondaries, while the S workload measures the data staleness on an arbitrary secondary node.

Figure 9 shows that when data staleness of a secondary copy exceeds the threshold set by the clients, the clients of Decongestant will not see this: Decongestant reacts in time and sends their reads to the primary. The workload used here is read-write TPC-C with S workload, on 60 clients. The blue horizontal dashed line shows the data staleness limit set by the clients, which is 10 seconds. The red squares in the figure depict the maximum data staleness of the secondaries, which sometimes goes beyond the threshold. However, the Decongestant clients are protected from this — all green circles are below their data staleness limit.

We have done a more detailed analysis on the MongoDB internal diagnostic data, trying to understand what happens behind this data staleness pattern (Figure 9). The high-level idea is that when the primary is overloaded, the secondaries trying to read the oplog from the primary can stall. That is, a secondary has a cursor open on the primary's oplog and calls the "getMore" operation. When the primary is overloaded, it can take a long time to service this operation, and during that time, the secondary can't move forward because it does not know about the newer operations (i.e., it gets more and more stale). Eventually the primary gets around to servicing the "getMore" request and sends a large batch of operations to the secondary. Since the secondary isn't overloaded, it can apply the operations quickly and catch up.

Now we discuss how the primary is stalled. During this period, several checkpoints completed on the primary and the disk of it was 100% utilised. The checkpoints took a long time to complete (around 30 seconds on average). During that time the latency grows. MongoDB had noticed the lag and was deliberately throttling update operations via a mechanism called "flow control" [27]. This might be part of the reason for the throughput of read-write TPC-C workload being unstable (Figure 4 (a)). Once the flush completes, the primary comes back to life and serves a batch of "getMore" operations quickly.

Our method to bound the data staleness still works reasonably well when the clients set the data staleness limit to a very low value, such as 3 seconds (see Figure 10). Such a low data staleness limit is challenging, as the granularity of the data staleness reported by MongoDB `serverStatus` is one second. So a system has little time to react to increasing staleness before the limit

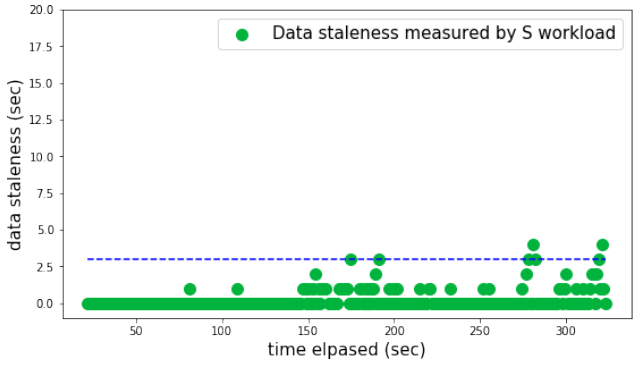

Figure 10: Data staleness measured by S workload in Decongestant with the data staleness limit set to be 3 seconds. The workload used here is read-write TPC-C with 200 clients.

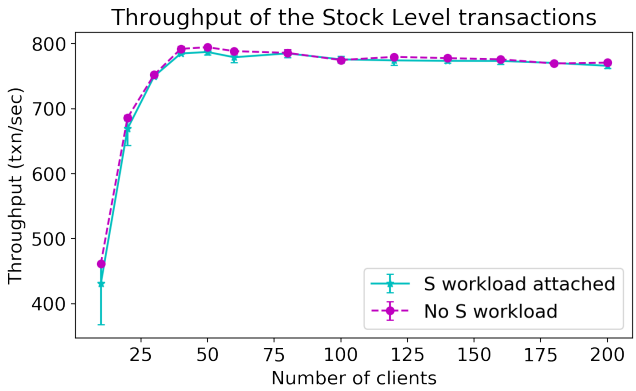

Figure 11: Throughput of Stock Level transactions when running the read-write TPC-C workload, with or without S workload.

is breached. The workload used in Figure 10 is read-write TPC-C with 200 clients. We can see, in most cases, the client-observed data staleness are bounded as requested, though two data points have staleness value 4 seconds.

4.6 Impact of S workload

Lastly, we claim that running S workload alongside a performance benchmark (such as read-write TPC-C), causes little distortion on the results recorded for performance. Figure 11 shows the throughput of Stock Level transactions, when running the read-write TPC-C workload with the S workload (as it does in our previous experiments), compared to what is measured when the read-write TPC workload is run alone. The Read Preference used here is Primary. We can see that the throughput of the performance benchmark remains at a similar level when running with or without the S workload.

5 RELATED WORK

Some text in this section already appeared in [21].

The trade-off between performance and consistency in distributed storage system has been studied for a very long time. This body of work is profoundly influenced by CAP theorem [9] and its later PACELC formulation [1].

Various storage systems offer different consistency properties. Some make the choice for the users, such as BigTable [10] and Spanner [13], which guarantee some form of strong consistency. Others give users some freedom to make their own decisions, including Amazon Dynamo [16], Cassandra [30], as well as MongoDB. There is also a trend that Database-as-a-Service is hosted on top of shared-disks systems (usually provided by large cloud computing vendors), such as Amazon Aurora [39] and Microsoft Socrates [3], whose features usually impact on the exposed consistency characteristics of these Database-as-a-Service.

There is a huge amount of work evaluating the behaviour of distributed storage systems, in order to help users make more informed choices. Wada et al and Bermbach et al benchmark a large variety of distributed storage systems from the customers' view [5–8, 40]. These works treat the distributed storage system as a black-box, and send read and write requests to it, just like normal customers would do. Our previous work [20] measured the inconsistency window between the primary copy and secondary copies of MongoDB Atlas at around 25 ms.

There are works which, rather than comparing the read and write results (as client-centric methods do), capture the trace of operations on various entities (chosen by the user), and then post-execution analysis determines whether an equivalent serial execution would give the same result in each read; if not an anomaly is reported [2, 17, 31].

Trading performance for data freshness for read-only transactions / queries has been explored [11, 18, 33]. For example [33] applied this idea to mix OLAP and OLTP workloads in a database cluster providing freshness guarantees, though requiring a central coordinator which sees all transactions. In contrast, our proposed Decongestant is decentralised, which can be used both by on-premises and cloud users, and can deduce overload situations by probing rather than seeing the complete workload.

Pileus is a self-configuring system based on a Service Level Agreement (SLA) [36]. Within one SLA, there are a few subSLAs. Each subSLA includes a consistency requirement, a latency bound, and a utility score. Similar to our work, Pileus has "monitors" residing on the client nodes (each client has one monitor), and these probe periodically to decide which node a reading request should be directed to, so that the highest utility score among all subSLAs is achieved. Tuba [4] is an extension for Pileus. Tuba is a DBMS which is able to reconfigure itself, based on the observed latency and subSLA hit and miss ratio from all clients. Possible reconfiguration includes: changing primary replica, adding or removing secondaries, and varying synchronization periods between the primary and secondary copies.

Some recent work for self-configuring database applies machine learning technologies. There are automated systems able to tune large number of database knobs [38], adding and deleting indices [14], forecasting workloads [32], scaling resources [15], providing advice on partitioning [19], etc.

6 CONCLUSION

We presented the design and evaluation of Decongestant, a system which is able to automatically and dynamically determine Read Preference settings for read operations in MongoDB, in order to get good performance while delivering fresh-enough data to clients. Our solution works for both on-premises MongoDB deployments and MongoDB-as-a-Service. The key innovation is a client-based way to detect when either the primary, or one of the secondaries, are congested, by comparing estimates of the

time taken on the relevant server for performing recent read operations. When congestion is detected through these estimates, the system shifts reads away from the congested server; however, this decision is also constrained by estimates of the current data staleness on the secondaries.

This design avoids the need in current practice for application programmers to hard-code the decision of whether reads should go to primary or to secondaries (and thus risk seeing stale values). Instead the decision is made dynamically at run time by Decongestant, adapting to the recent situation in the servers.

Our experimental evaluation is done with YCSB-A, YCSB-B, and with workloads that run TPC-C transactions with a balance between read-only and updating transactions. We showed that Decongestant is able to adapt to workload shifts as they occur, and that it delivers good performance that respects client-chosen limits on data staleness. Indeed, in read-intensive workloads such as YCSB-B, we can outperform both hard-coded alternatives.

In future work, we plan to look at more sophisticated feedback control when adjusting read preference, and to support richer client SLAs as well as maximum staleness. We will explore the possibility of extending Decongestant to other database systems, which have a leader-follower architecture similar to MongoDB.

ACKNOWLEDGMENTS

This work was supported by the Australian Research Council (ARC) Linkage Project LP160100883.

REFERENCES

[1] Daniel Abadi. 2012. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *Computer* 45, 2 (2012), 37–42.

[2] Yazeed Alabdulkarim, Marwan Almaymoni, and Shahram Ghandeharizadeh. 2017. *Polygraph.* Technical Report 2017-02. Database Laboratory, Computer Science Department, University of Southern California.

[3] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, et al. 2019. Socrates: The New SQL Server in the Cloud. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD'19)*. 1743–1756.

[4] Masoud Saeida Ardekani and Douglas B Terry. 2014. A self-configurable geo-replicated cloud storage system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*. 367–381.

[5] David Bermbach. 2014. *Benchmarking eventually consistent distributed storage systems.* KIT Scientific Publishing Karlsruhe.

[6] David Bermbach and Stefan Tai. 2011. Eventual consistency: How soon is eventual? An evaluation of Amazon S3's consistency behavior. In *Proceedings of 6th Workshop on Middleware for Service Oriented Computing*. ACM, 1.

[7] David Bermbach and Stefan Tai. 2014. Benchmarking eventual consistency: Lessons learned from long-term experimental studies. In *IEEE International Conference on Cloud Engineering (IC2E'14)*. 47–56.

[8] David Bermbach, Erik Wittern, and Stefan Tai. 2017. *Cloud service benchmarking*. Springer.

[9] Eric A Brewer. 2000. Towards robust distributed systems. In *PODC*. 7.

[10] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems* 26, 2 (2008), 4.

[11] James Cipar, Greg Ganger, Kimberly Keeton, Charles B. Morrey, III, Craig A.N. Soules, and Alistair Veitch. 2012. LazyBase: Trading Freshness for Performance in a Scalable Database. In *Proceedings of 7th ACM European Conference on Computer Systems (EuroSys'12)*. 169–182.

[12] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC'10)*. 143.

[13] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems* 31, 3 (2013), 8.

[14] Sudipto Das, Miroslav Grbic, Igor Ilic, Isidora Jovandic, Andrija Jovanovic, Vivek R Narasayya, Miodrag Radulovic, Maja Stikic, Gaoxiang Xu, and Surajit Chaudhuri. 2019. Automatically indexing millions of databases in Microsoft Azure SQL database. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD'19)*. 666–679.

[15] Sudipto Das, Feng Li, Vivek R Narasayya, and Arnd Christian König. 2016. Automated demand-driven resource scaling in relational database-as-a-service. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD'16)*. 1923–1934.

[16] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007 (SOSP 2007)*. ACM, 205–220.

[17] Wojciech Golab, Muntasir Raihan Rahman, Alvin AuYoung, Kimberly Keeton, and Indranil Gupta. 2014. Client-centric benchmarking of eventual consistency for cloud storage systems. In *IEEE 34th International Conference on Distributed Computing Systems (ICDCS'14)*. 493–502.

[18] Hongfei Guo, Per-Åke Larson, and Raghu Ramakrishnan. 2005. Caching with 'Good Enough' Currency, Consistency, and Completeness. In *Proceedings of 31st International Conference on Very Large Data Bases (VLDB'05)*. 457–468.

[19] Benjamin Hilprecht, Carsten Binnig, and Uwe Röhm. 2019. Towards learning a partitioning advisor with deep reinforcement learning. In *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*. ACM, 6.

[20] Chenhao Huang, Michael Cahill, Alan Fekete, and Uwe Röhm. 2018. Data Consistency Properties of Document Store as a Service (DSaaS): Using MongoDB Atlas as an Example. In *Technology Conference on Performance Evaluation and Benchmarking (LNCS 11135)*. Springer, 126–139.

[21] Chenhao Huang, Michael Cahill, Alan Fekete, and Uwe Röhm. 2020. Deciding When to Trade Data Freshness for Performance in MongoDB-as-a-Service. In *IEEE 36th International Conference on Data Engineering (ICDE'20)*. 1934–1937.

[22] MongoDB Inc. 2020. MongoDB Documentation: Sharding. https://docs.mongodb.com/manual/sharding/. Accessed:2020-05-06.

[23] MongoDB Inc. 2020. Read Concern - MongoDB Manual. https://docs.mongodb.com/manual/reference/read-concern/. Accessed: 2020-05-06.

[24] MongoDB Inc. 2020. Read Preference - MongoDB Manual. https://docs.mongodb.com/manual/core/read-preference/. Accessed: 2020-05-06.

[25] MongoDB Inc. 2020. Read Preference maxStalenessSeconds. https://docs.mongodb.com/manual/core/read-preference-staleness/#replica-set-read-preference-max-staleness. Accessed: 2020-06-19.

[26] MongoDB Inc. 2020. Read Preference Use Cases. https://docs.mongodb.com/manual/core/read-preference-use-cases/. Accessed: 2020-10-13.

[27] MongoDB Inc. 2020. Replication. https://docs.mongodb.com/manual/replication/#replication-lag-and-flow-control. Accessed: 2020-05-06.

[28] MongoDB Inc. 2020. Server Selection Algorithm. https://docs.mongodb.com/manual/core/read-preference-mechanics/. Accessed: 2020-06-19.

[29] Asya Kamsky. 2019. Adapting TPC-C Benchmark to Measure Performance of Multi-Document Transactions in MongoDB. *PVLDB* 12, 12 (2019), 2254–2262.

[30] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40.

[31] Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. 2015. Existential consistency: measuring and understanding consistency at Facebook. In *Proceedings of 25th ACM Symposium on Operating Systems Principles (SOSP'15)*. 295–310.

[32] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J Gordon. 2018. Query-based workload forecasting for self-driving database management systems. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD'18)*. 631–645.

[33] Uwe Röhm, Klemens Böhm, Hans-Jörg Schek, and Heiko Schuldt. 2002. FAS – A Freshness-Sensitive Coordination Middleware for a Cluster of OLAP Components. In *Proceedings of 28th International Conference on Very Large Data Bases (VLDB'02)*. 754–765.

[34] Fred B Schneider. 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *Comput. Surveys* 22, 4 (1990), 299–319.

[35] William Schultz, Tess Avitabile, and Alyson Cabral. 2019. Tunable Consistency in MongoDB. *PVLDB* 12, 12 (2019), 2071–2081.

[36] Douglas B Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K Aguilera, and Hussam Abu-Libdeh. 2013. Consistency-based service level agreements for cloud storage. In *Proceedings of 24th ACM Symposium on Operating Systems Principles (SOSP'13)*. ACM, 309–324.

[37] Misha Tyulenev, Andy Schwerin, Asya Kamsky, Randolph Tan, Alyson Cabral, and Jack Mulrow. 2019. Implementation of Cluster-wide Logical Clock and Causal Consistency in MongoDB. In *Proceedings of ACM International Conference on Management of Data, (SIGMOD'19)*. 636–650.

[38] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. 2017. Automatic database management system tuning through large-scale machine learning. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD'17)*. 1009–1024.

[39] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD'17)*. 1041.

[40] Hiroshi Wada, Alan Fekete, Liang Zhao, Kevin Lee, and Anna Liu. 2011. Data Consistency Properties and the Trade-offs in Commercial Cloud Storage: the Consumers' Perspective.. In *CIDR'11*, Vol. 11. 134–143.