Deciding When to Trade Data Freshness for Performance in MongoDB-as-a-Service

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*—MongoDB is a popular document store that is also available as a cloud-hosted service. MongoDB internally deploys primary-copy asynchronous replication, and it allows clients to vary the *Read Preference*, so reads can deliberately be directed to secondaries rather than the primary site. Doing this can sometimes improve performance, but the returned data might be stale, whereas the primary always returns the freshest data value. While state-of-practice is for programmers to decide where to direct the reads at application development time, they do not have full understanding then of workload or hardware capacity. It should be better to choose the appropriate Read Preference setting at runtime, as we describe in this paper.**

We show how a system can detect when the primary copy is saturated in MongoDB-as-a-Service, and use this to choose where reads should be done to improve overall performance. Our approach is aimed at a cloud-consumer; it assumes access to only the limited diagnostic data provided to clients of the hosted service.

***Index Terms*—Database-as-a-Service, performance, replication**

I. Introduction

Databases-as-a-Service (DBaaS) have become a popular cloud offering due to their scalability, elasticity, and the reduced total cost of ownership. DBaaS services typically deploy a combination of data sharding and replication over multiple nodes to provide a good balance of scalability and availability. This means that there are typically multiple copies of data available, and some DBaaS 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 in a dynamic workload environment while minimizing its pitfalls, namely access to potentially stale data. One typical example in this space is MongoDB, a popular NoSQL database management system. MongoDB Atlas[1] is the corresponding cloud-hosted MongoDB service.

MongoDB internally uses asynchronous primary-copy replication for fault-tolerance. It is usually run as a replica set, where each node keeps a logical copy of the database [1]. Each replica set is a log-replicated state machine [2], and all nodes are usually placed within one geographical region. There is one primary copy which processes all write operations. Each secondary copy pulls the updated logs from the primary and then replays it to keep up with the primary. This means the data on the secondary copies might be stale compared to the data on the primary copy. 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 [3]). When reading from the primary copy on a healthy cluster with default settings, fresh data is returned. Nevertheless, in situations where the primary copy is saturated, the read latency can be huge. If a user wants a loaded system to have larger throughput and lower read latency, s/he can choose to read from secondary copies; however, data returned might thus be stale.

The state-of-the-art practice is to "hard-code" the Read Preference. A major problem with this approach is that developers have insufficient information to make the best decision when the code is written. The "best" Read Preference choice can change at runtime depending on workload.

This paper shows how to capture knowledge at runtime of the current condition, so the Read Preference can be adjusted to suit. There are a few challenges to reach this goal. Firstly, unlike in traditional server deployments, the clients in a cloud-based system lack much of the information about operating statistics. A second challenge is to make the solution independent from the DBMS's hardware, DBMS's configurations, and workload. We want the system to make local decisions for a client with only local knowledge and low overheads.

Here, we describe how to use the read transaction latency measured by the clients, to judge whether the primary is saturated and the secondaries have capacity, in which case reads should be shifted to the secondaries. We have a preliminary implementation of a component called Read Preference Manager, that does this.

II. Design of the Read Preference Manager

A simplified architecture of our proposal is shown in Figure 1. A component called Read Preference Manager resides on the client system where multiple client applications are executing. The Read Preference Manager is a service that makes suggestions regarding primary or secondary in order to improve overall performance. It does so by keeping track of the access latencies experienced by the clients. In order to be able to compare latencies of both choices (primary

This research forms part of the Australian Research Council (ARC) Linkage Project LP160100883.

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

2375-026X/20/$31.00 ©2020 IEEE
DOI 10.1109/ICDE48307.2020.00207

or secondary), the protocol is such that it sends a client with a certain percentage to the opposite choice in order to gather latency evidence from both primary and secondary. In our implementation, Read Preference Manager is coded using the Python multiprocessing module, and it shares some state variables with the clients: two queues which keep transaction latencies (for reads to Primary and Secondaries, respectively); also the most recent recommendation from the Manager.

A client must be written to communicate with the Read Preference Manager, as follows. Before invoking any read-only transaction call, the client examines the shared variable with the most recent recommendation from the Read Preference Manager. Mostly, the client follows the suggestion, by using the suggested Read Preference in the invocation. However, for a small proportion (called the *Exploration Fraction*) of read-only invocations, they are directed to the opposite choice from what is suggested by the Read Preference Manager. These invocations are chosen as volunteers to ensure adequate data for the Manager's suggestions. The clients keep track of the latency of their read-only transactions, and report them to the Read Preference Manager through one of the shared queues (depending on which Read Preference was actually used in the call).

The Read Preference Manager runs once per minute in our implementation. On system startup, the Read Preference Manager initializes the Read Preference suggestion as Primary. After each period, the Manager retrieves 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 empties the queue of latencies for read-only transactions sent to the secondaries, and determines the median of those. The Read Preference Manager uses the ratio of the median latency of reads on secondaries, to the median latency of reads on the primary. When this ratio is below a threshold, called *Decision Threshold*, then the recommendation made by the Read Preference Manager would be Secondary, because this suggests that the primary is being saturated while the secondaries have capacity. On the other hand, if the ratio of median latencies is more than the same threshold, the suggestion of the Read Preference Manager would be Primary.

III. Experiments

In the following, we present the results of experiments showing that the Read Preference Manager does trade data freshness for performance sensibly, with MongoDB Atlas. The metrics are throughput, 80-percentile latency, and data freshness. For these experiments, we use the transactions from TPC-C [4], with 100 warehouses and scaling factor 1. Unlike the standard workload, we increase the relative frequency of Stock Level transactions to 50%, because the impact of directing the read-only transactions to the secondary copies is more obvious when the percentage of read-only transactions are large. We run the implementation of TPC-C transactions by Kamsky [5] who adapted the benchmark to the MongoDB query language, transaction semantics, and coding practices.

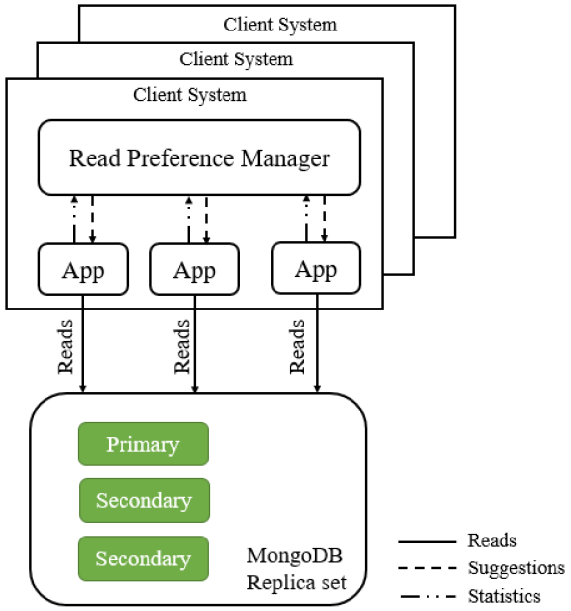

Fig. 1. A simplified architecture of the Read Preference Manager

We consider three systems: one where our Read Preference Manager is used, and two others which hard-code the Read Preference for read-only transactions, either to Primary or to Secondary.

The experiments were executed in the AWS cloud, with the TPC-C clients on a AWS c4.4xlarge instance which has 16 vCPUs and 30 GB RAM, connecting to a MongoDB Atlas instance in the same data center (region ap-southeast-2) on a M60 Low CPU, with 61GB RAM and 8 vCPUs. The reported results are averages over 5 runs, and in all figures the plots show error bars from the minimum to maximum measured values.

We set the Read Preference Manager's *Exploration Fraction* value to 10%, which means that 10% of the clients are serving as volunteers to explore different options for the whole system. The *Decision Threshold* value chosen is 50%. This denotes that if the 50-percentile (median) latency of all Primary reading requests is double the median of all Secondary reading requests in the previous period, then the Read Preference Manager will suggest the Read Preference to be Secondary in the next round.

Figure 2 shows the throughput of the Stock Level transactions[2]. When the Read Preference Manager is active and the number of clients is between 10 and 30, the Manager always recommends Primary (see Figure 5), and throughput of the Stock Level transaction is a bit lower than the one when Primary is a hard-coded preference, showing some small overhead in the Manager. The Read Preference Manager is redirecting some read-only transactions when the number of

[2]While all TPC-C transactions are executed in this experiment, only the throughput of the Stock Level transaction is reported.

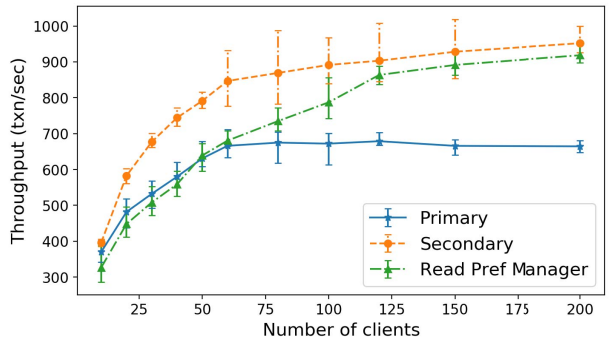

Fig. 2. Throughput of the Stock Level transactions with a read-intensive TPC-C workload (50% Stock Level transactions) and activated Read Preference Manager.

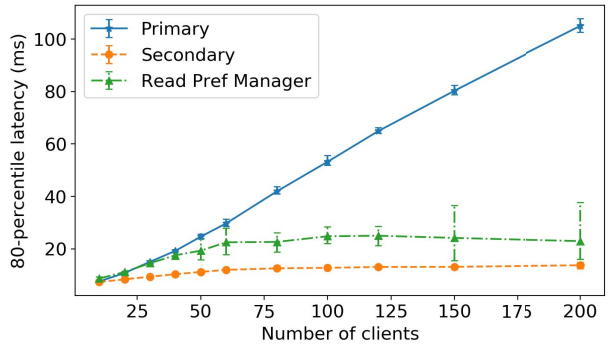

Fig. 3. 80-percentile latency of Stock Level transactions with read-intensive TPC-C workload and activated Read Preference Manager.

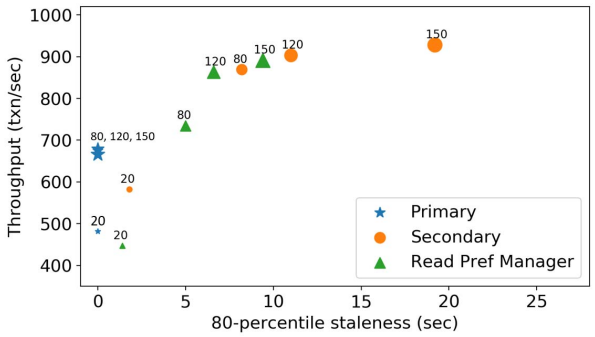

Fig. 4. The trade-off between the throughput and 80-percentile staleness of the Stock Level transactions. The number above the marks and the size of them indicates the number of clients.

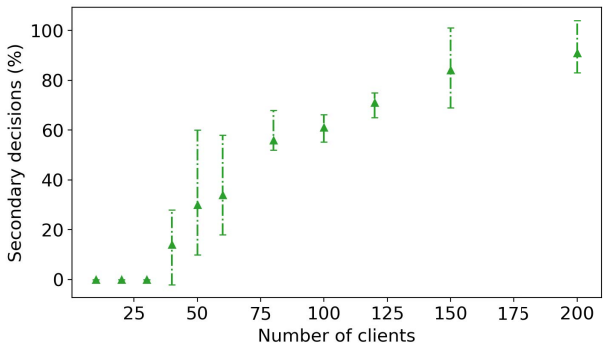

Fig. 5. Percentage of times when Read Preference Manager decides Read Preference to be Secondary.

clients is larger than 40, and the throughput of the Stock Level transactions catches up to, and then exceeds, what would happen when Stock Level transactions hard-coded to the primary. With 120 clients or more, the throughput of the Stock Level transactions approaches the same level as when all of them are hard-coded to the secondaries. But it never surpasses or equals to it, even when most recommendations of the Manager are Secondary. The reason is two-fold: Firstly, the Stock Level transactions are still sent to the primary copy for the first minute of the 10-minute long run. Also, we have 10% of volunteers doing the exploration work for the system and so using the sub-optimal selection.

Figure 3 shows the 80-percentile latency of the Stock Level transaction. With the help of the Read Preference Manager, the 80-percentile latency of the Stock level transaction is generally stable - it does not keep increasing as happens when Primary is hard-coded as preference.

As a way of evaluating data freshness, we report *Replication Lag* [6] from MongoDB's internal diagnostic data. *Replication Lag* is the delay between an operation on the primary copy and the application of that operation on the secondary copies, as recorded by each secondary copy.

Figure 4 demonstrates the trade-off between the throughput of the Stock Level transactions and the 80-percentile data staleness. The number above the marks and the size of them shows the number of clients. From this Figure, we can see that when the client numbers are small, such as 20 clients, using Read Preference Manager loses some throughput and data freshness compared to always directing the Stock Level transactions to the primary copy. However, when the the client number is 120 and 150, the benefit of using Read Preference Manager is clear, offering a sweet spot in terms of throughput and 80-percentile staleness. For example, when the client number is 120, the throughput of the Stock Level transactions is around 30% higher than when they are all directed to the primary. The data staleness is around 50% less than when all of them are directed to the Secondaries.

Figure 5 shows the percentage of times when the suggested Read Preference is Secondary, over a varying number of clients. We can see that with a load of between 10 to 30 clients, the recommendation made by the Read Preference Manager is always Primary. The probability of suggesting Read Preference as Secondary then gradually goes up. At 200 clients, Secondary is suggested most of the time.

IV. Related Work

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 [7] and its later PACELC formulation [8].

Various storage systems address this. Some systems make the choice for the users, such as BigTable [9] and Spanner [10], which guarantee some form of strong consistency. Others give users some freedom to make their own decisions, including Amazon Dynamo [11], Cassandra [12], as well as MongoDB.

There is a huge amount of work measuring 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 [13]–[17]. Our previous work [18] measured the inconsistency window between the primary copy and secondary copies of MongoDB Atlas. We found that under the conditions measured, the window is around 25 ms.

The idea of trading performance for data freshness for read-only transactions / queries has been explored before [19]–[21]. For example [19] applied this idea to mix OLAP and OLTP workloads in a database cluster and even providing freshness guarantees, though requiring a central coordinator which sees all transactions. In contrast, our proposed Read Preference Manager is decentralised, aiming at a cloud-consumer, and can deduce overload situations by probing rather than requiring to see the complete workload.

Pileus [22] is a self-configuring system based on a Service Level Agreement (SLA), within which are subSLAs. Each subSLA includes a consistency requirement, a latency bound, and a utility score. Pileus "monitors" reside on the clients, and probe periodically to decide which node a reading requests should be directed to, so that the highest utility score among all subSLAs is achieved. In Pileus each client has its own monitor, whereas we share the work among clients on a site. Tuba [23] extends this idea and is a database management system 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.

V. Conclusions

We presented the preliminary design of a Read Preference Manager for MongoDB Atlas which is capable of automatically choosing the appropriate Read Preference for read-only transactions at runtime in response to the current workload, avoiding that developers have to make this performance decision statically during coding time. The Read Preference Manager allows transactions to benefit from increased performance (though with reduced freshness for readers) by reading from secondaries when a database cluster is heavily loaded. When load is reduced, and performance won't suffer, the Read Preference Manager guides the client requests to the primary, for better data freshness. Our experiments with a read intensive TPC-C benchmark showed that the Read Preference Manager can indeed trade data freshness for performance. Further studies are planned with other benchmarks and settings.

References

[1] W. Schultz, T. Avitabile, and A. Cabral, "Tunable consistency in mongodb," *PVLDB*, vol. 12, no. 12, pp. 2071–2081, 2019.

[2] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys (CSUR)*, vol. 22, no. 4, pp. 299–319, 1990.

[3] "Read preference - mongodb manual," https://docs.mongodb.com/manual/core/read-preference/, accessed: 2019-08-26.

[4] "Tpc-c benchmark," http://www.tpc.org/tpcc/.

[5] A. Kamsky, "Adapting tpc-c benchmark to measure performance of multi-document transactions in mongodb," *PVLDB*, vol. 12, no. 12, pp. 2254–2262, 2019.

[6] "Mongodb documentation: Troubleshoot replica sets: Check the replication lag," https://docs.mongodb.com/manual/tutorial/troubleshoot-replica-sets/#check-the-replication-lag, accessed:2019-10-13.

[7] E. A. Brewer, "Towards robust distributed systems," in *PODC*, 2000.

[8] D. Abadi, "Consistency tradeoffs in modern distributed database system design: Cap is only part of the story," *Computer*, vol. 45, no. 2, pp. 37–42, 2012.

[9] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[10] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild *et al.*, "Spanner: Google's globally distributed database," *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 3, p. 8, 2013.

[11] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," in *Proceedings of SOSP*, 2007, pp. 205–220.

[12] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[13] H. Wada, A. Fekete, L. Zhao, K. Lee, and A. Liu, "Data consistency properties and the trade-offs in commercial cloud storage: the consumers' perspective." in *CIDR*, 2011, pp. 134–143.

[14] D. Bermbach and S. Tai, "Eventual consistency: How soon is eventual? an evaluation of amazon s3's consistency behavior," in *Proceedings of the 6th Workshop on Middleware for Service Oriented Computing*. ACM, 2011, p. 1.

[15] D. Bermbach, *Benchmarking eventually consistent distributed storage systems*. KIT Scientific Publishing Karlsruhe, 2014.

[16] D. Bermbach and S. Tai, "Benchmarking eventual consistency: Lessons learned from long-term experimental studies," in *Proceedings of IEEE Conference on Cloud Engineering (IC2E)*, 2014, pp. 47–56.

[17] D. Bermbach, E. Wittern, and S. Tai, *Cloud service benchmarking*. Springer, 2017.

[18] C. Huang, M. Cahill, A. Fekete, and U. Röhm, "Data consistency properties of document store as a service (DSaaS): Using MongoDB Atlas as an example," in *Technology Conference on Performance Evaluation and Benchmarking (Springer LNCS 11135)*, 2018, pp. 126–139.

[19] U. Röhm, K. Böhm, H. Schek, and H. Schuldt, "FAS – A freshness-sensitive coordination middleware for a cluster of OLAP components," in *Proceedings of VLDB 2002*, 2002, pp. 754–765.

[20] H. Guo, P. Larson, and R. Ramakrishnan, "Caching with 'good enough' currency, consistency, and completeness," in *Proceedings of VLDB*, 2005, pp. 457–468.

[21] J. Cipar, G. Ganger, K. Keeton, C. B. Morrey, III, C. A. Soules, and A. Veitch, "Lazybase: Trading freshness for performance in a scalable database," in *Proceedings of EuroSys*, 2012, pp. 169–182.

[22] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh, "Consistency-based service level agreements for cloud storage," in *Proceedings of SOSP*. ACM, 2013, pp. 309–324.

[23] M. S. Ardekani and D. B. Terry, "A self-configurable geo-replicated cloud storage system," in *Proceedings of USENIX OSDI*, 2014, pp. 367–381.