DeathStar Movie for Geo-Distributed Databases

Stressing databases using a movie review site

Samuel van den Houten[1]

Supervisors: Dr. Asterios Katsifodimos[1]**, Oto Mráz**[1]

[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 22, 2025

Name of the student: Samuel van den Houten
Final project course: CSE3000 Research Project
Thesis committee: Dr. Asterios Katsifodimos, Oto Mráz, Prof.dr. Koen Langendoen

Abstract

Geo-distributed databases offer the scalability and low latency that contemporary applications demand, but are challenging to implement. It is therefore crucial that they are tested well. Established benchmarks, such as TPC-C and YCSB-T, are limited and do not cover the entire set of workloads that geo-distributed databases are subjected to. In this paper, we discuss the adaptation of DeathStar Movie - an existing benchmark for microservice systems - for benchmarking these geo-distributed databases. We set up an experiment in which we ran this modified version of the benchmark on four database systems: Detock, SLOG, Calvin, and Janus. The results showed that DeathStar Movie is capable of pushing these systems to their limits. It also showed the different performance characteristics of the systems, stemming from the different ways in which they attempt to overcome the challenges posed by a geo-distributed setting.

1 Introduction

Geo-distributed databases are crucial for today's large-scale, worldwide applications. They are already used by large players within the industry to achieve high scalability and low latency [1] [2]. Geo-distributed databases are also useful for their ability to store data regionally, as this allows for compliance with recent legislation dictating the storage location of user data [3]. However, implementation of these systems is challenging, especially when considering ACID transactions and strict serializability. Common approaches can lead to high abort rates or deadlock, hurting performance [4]. It is therefore highly important that these geo-distributed database systems are tested well.

In the present, geo-distributed databases are mostly tested using industry-standard benchmarks like TPC-C [5] and YCSB-T [6]. However, these benchmarks are limited in scope and not capable of exposing all of the potential choke points of geo-distributed databases [7]. TPC-C in particular was first released in 1992, which means that the database systems it was designed to test are quite different from the state-of-the-art.

Recognizing the need for additional benchmarking tools, we propose the use of the DeathStar Movie benchmark [8] for geo-distributed databases. This is an existing benchmark originally developed for testing microservice systems, which can be modified to target databases specifically. The main research question for this paper is therefore: "How do geo-distributed databases perform on the DeathStar Movie benchmark?".

This paper makes the following contributions:

- A modification of DeathStar Movie for benchmarking geo-distributed databases, which extracts the underlying database transactions from the original workload.
- An implementation of this modified design within the codebase of Detock, a geo-distributed database
- An experiment in which we test six variations ("scenarios") of this implementation using four geo-distributed database systems.

This paper proceeds as follows: In section 2 we will start by providing background information on DeathStar Movie and on the database systems we used for testing. In section 3 we follow up with a description of the modifications we made to make DeathStar Movie applicable to distributed databases. Section 4 contains a technical description of the experiment along with the main results. In section 5 we reflect on the ethical implications and reproducibility of the methods used. Section 6 follows with describing the limitations of this paper, comparisons with other research, and recommendations for future work. In section 7 we end with the main conclusions.

2 Background

This section includes background information on the benchmark and selected database systems. Section 2.1 starts by describing the original DeathStar Movie benchmark. In Section 2.2 we describe the database systems and protocols .

2.1 DeathStar Movie

DeathStar Movie is a benchmark for cloud microservices. It is part of the larger DeathStarBench suite. It was first described in 2019 by Gan Yu et al. [8], though this paper considers an updated version found on Github [9]. DeathStar Movie models a movie database and review website, akin to *IMDB* or *Letterboxd*.

DeathStar Movie sets up a network of microservices and MongoDB databases, along with Redis and Memcached caches. A visualization of this can be seen in Figure 1. The system allows movies to be added, along with their plot descriptions and casts. Users can login and submit reviews. The workload for this benchmark consists of repeatedly submitting reviews through an HTTP endpoint, which are then written to the databases and caches. This workload can be found in the *wrk* directory of the DeathStar Movie repository [9].

While DeathStar Movie's utility for evaluating cloud microservice systems is known, its use as a benchmark purely for (geo-distributed) databases has not been extensively explored. It being the subject of this research therefore helps in understanding the current landscape of benchmarks and could justify its use alongside industry-standard benchmarks like TPC-C.

2.2 Evaluated database systems

In this paper we evaluate DeathStar Movie with several geo-distributed database systems: Detock, SLOG, Janus, and Calvin. We use these different systems because they each have a different protocol for handling transaction conflicts. This has been shown to affect their performance characteristics in different scenarios [10] [4].

Calvin was introduced in 2012 in a paper by Alexander Thompson et al. [11] Its goal is to be a scalable distributed database with high availability and ACID transactions. It allows for multiple replicas of database partitions,

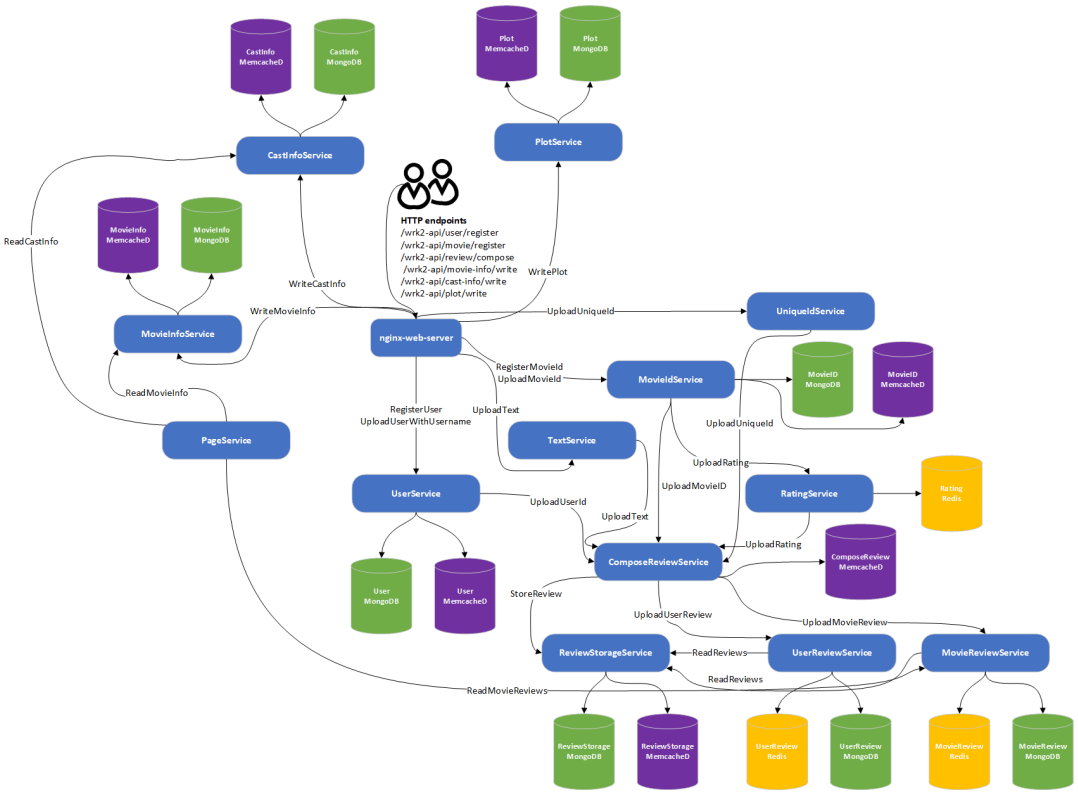

Figure 1: DeathStar Movie architecture
Source: [9]

which are kept consistent by globally ordering every transaction. Servers accumulate requests during 10ms epochs, after which batches of transactions are shared with other servers and executed in a deterministic order.

Janus is a protocol introduced by Shuai Mu et al. [12] in 2016. While some database systems use separate protocols for concurrency control and generating consensus, Janus consolidates these into one protocol. This makes it perform particularly well when encountering high contention.

SLOG is a database system first described by Kun Ren et al. [13] in 2019. It achieves much lower latency than preceding systems like Calvin, while retaining high throughput. It assigns a ”home” region to every record, which allows transactions involving only a single home to skip the global ordering and be resolved faster.

Detock is a database system introduced in a 2023 paper by Cuong D. T. Nguyen et al. [4] It improves on SLOG by eliminating its global ordering layer entirely, reducing latency. Instead, it uses a new graph-based algorithm for deterministically scheduling the transactions at each region. This means that the ordering graph will be created in the same way in every region, with just the transaction data and no further communication.

3 Adapting Deathstar Movie for geo-distributed databases

In order to apply DeathStar Movie to geo-distributed databases we needed to apply several changes to the original benchmark. Firstly, we removed the microservices (blue in Figure 1) and instead implemented the underlying transaction directly. This eliminates the possibility of a bottleneck and focuses the benchmark solely on the database. We removed the Redis and Memcached caches (yellow and purple in the figure) for the same reasons. Since we no longer use microservices, we can combine the eight MongoDB databases (green in the figure) into a single relational database with multiple tables. While we could convert the databases to tables one-to-one, we made some optimizations in this step. Some databases, like *CastInfo*, are not used by the workload at all and can therefore be removed. Some databases, like *ReviewStorage*, *UserReview*, and *MovieReview* store redundant data. While this is useful for quick access for their corresponding microservices, this redundancy is not needed when using a single database. The final database schema resulting from this process can be seen in Figure 2.

The original benchmark stores both a database that stores just the submitted review and another that stores user records which each have an array with their submitted reviews. Adapting the benchmark into a relational schema means it would be logical to only store a *Reviews* table. However, we chose to keep storing the reviews with the user records as

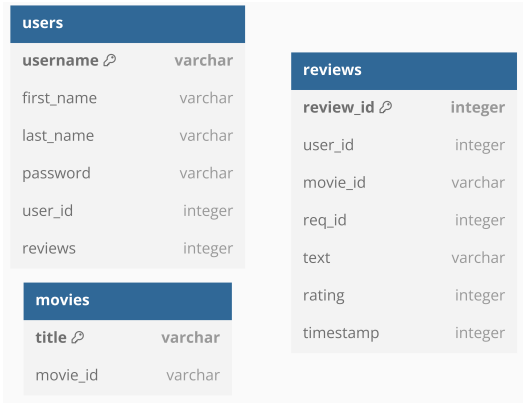

Figure 2: Adapted database schema

well, since this increases transaction conflicts and therefore strain on the database. However, since some of the databases tested do not allow arrays as a datatype, this is instead implemented as a counter that is raised every time a user submits a review. The transaction that is used for the modified DeathStar Movie, which we have named $NewOrder$, is described by the following SQL code:

```
INSERT INTO reviews (user_id, movie_id, req_id,
text, rating, timestamp, review_id)
VALUES (
  (SELECT user_id FROM users
  WHERE username = <USERNAME>),
  (SELECT movie_id FROM movies
  WHERE title = <TITLE>),
  <REQ_ID>,
  <TEXT>,
  <RATING>,
  <TIMESTAMP>,
  <REVIEW_ID>
);

UPDATE users
SET reviews = reviews + 1
WHERE username = <USERNAME>;
```

4 Experimental Setup and Results

This section documents the technical details of the experiment as well as the results. Section 4.1 describes the general setup we used. Section 4.2 follows with a description of the experiment and the metrics. Sections 4.3 through 4.8 present the various scenarios and their results.

4.1 General setup

The benchmark was run on a TU Delft cluster consisting of 4 machines. Each machine was equipped with 2× AMD EPYC 7H12 CPUs (128 cores, 256 threads total), along with 512GB of DDR4 RAM. The databases and benchmarks were run within Docker containers based on Ubuntu 22.04. The databases were configured to have 2 database regions and 2 partitions, which allowed for 2 machines per region. The round-trip latency between the different machines can be seen in Table 1.

	R0P1	R1P0	R1P1
R0P0	0.096	0.212	0.186
R0P1	0.112	0.151	
R1P0	0.097		

Table 1: Round-trip time between all configurations of regions R and partitions P (ms)

The benchmark framework allows the user to specify the number of (virtual) clients simultaneously submitting transactions. It also allows them to specify the fraction of transactions that should interact with multiple database regions and multiple database partitions. Unless specified otherwise, the below benchmarks use 3000 virtual clients, 50% multi-region transactions and 50% multi-partition transactions.

Prior to the workload, the databases are filled with 1000 users and 1000 movies [9], following the original benchmark. These records are distributed evenly across the different database partitions and regions. During the workload, transactions are randomly generated from valid users and movies, along with a randomly generated string for the review. Unless stated otherwise, transactions are submitted concurrently from 3000 virtual clients.

4.2 Experiment

The experiment consists of running the benchmark on several geo-distributed database systems: Detock [4], Slog [13], Janus [12] and Calvin [11]. We used these systems because they have different ways of handling transaction conflicts. This means that their performance could be affected differently by changing scenarios [10].

The benchmark results are evaluated using the following metrics:

- **Throughput** - Amount of transactions handled per second
- **Latency** - Mean latency for transactions
- **Bytes transferred** - Total amount of bytes transferred over the network
- **Cost** - Hypothetical cost in USD to run this experiment on a commercial cloud network for 1 hour.

The *Cost* metric is calculated using the following formula:

$$Cost = N * A + \frac{0.02G}{D} * 3600$$

Where:

- N is the number of machines used in the benchmark
- A is the average hourly cost of deploying a m4.2xlarge VM across 8 AWS regions: euw1, euw2, usw1, usw2, use1 use2, apne1 and apne2
- G is the amount of data transferred between database regions in GB
- D is the duration of the benchmark in seconds

We used a cost of 0.02$/GB as this is a common base rate for inter-region transfers on AWS [14]. The experiment is run on a compute cluster in order to provide a realistic scenario for a geo-distributed database.

4.3 Baseline

The baseline scenario tests the effect of increasing the amount of transactions that involve data from multiple database regions. Because of the additional coordination needed, these transactions put more strain on the system. This scenario is controlled by the *mh_chance*, which represents a percentage of transactions that interact with data from different home regions. The three records in a single-home transaction (a $User$, $Movie$, and $Review$) will always come from the same region as the server receiving the transaction. A multi-home transaction will utilize a $User$ from the local region, while the $Movie$ and $Review$ will be from another region. This results in database writes to 2 different regions. The other scenarios all use a constant mh_chance of 50%. All scenarios also use a mp_chance of 50%, which works similarly to mh_chance but instead controls the amount of transactions utilizing records from multiple database partitions.

The results for the baseline scenario can be seen in Figure 3. Note that while the P50, P95 and P99 latencies are shown for most systems, only P50 is shown for Janus and only P50 and P95 for Calvin. This is because the latter metrics had large outliers, up to 50000ms. The results are generally as expected, with the throughput of SLOG and Detock going down as the fraction of multi-home transactions increases. At very high multi-home percentages they are even outperformed by Calvin, which is unaffected due to its lack of a region system. We also observed Calvin transferring less data for the same throughput, therefore also making it more cost-effective. This trend continues in the other scenarios. We observed spikes in latency for SLOG at lower mh_chance, though since these appear erratically and only in the P99 and (partially) P95, these are assumed to be outliers.

4.4 Skew

In the skew scenario, data access is skewed, meaning some records are accessed more than others. The benchmark picks the records used in a transaction by generating a list of valid IDs based on the target home and partition. By default, a uniform random generator is used to index this list and get the ID. The skew scenario uses a non-uniform random generator instead, using the following formula:

$$Index = (A \mid B) \mod M + 1$$

Where:

A	$\sim \mathcal{U}(0, Skew * M)$
B	$\sim \mathcal{U}(0, M)$
M	is the maximum index for the ID candidate list
$Skew$	is the skewness factor controlled by the benchmark

The results can be seen in Figure 4. As with the baseline scenario, the P95 and P99 latencies consist mostly of outliers, while the P50 latencies are generally constant. When looking at the throughput, we can see that the skew factor has a minimal effect on SLOG, Detock and Calvin. However, it does have a large effect on Janus, which goes from 30000 txn/s to just 10000 txn/s when increasing the skew factor from 0.0 to 0.1. This suggests that Janus performs poorly when under high contention.

4.5 Sunflower

In this scenario, the data access is skewed towards records owned by a certain database region. While it could seem natural for a benchmark to equally distribute the load over the database regions, this is not realistic for geo-distributed databases. As users are more active during the day, this is when database traffic will be the heaviest. Due to differing time zones, this can lead to transactions being skewed towards certain regions. This pattern is simulated in the *Sunflower* scenario. This scenario is controlled by two parameters, $sunflower\_home$ and $sunflower\_chance$. The $sunflower\_chance$ parameter is a percentage value that controls the odds that the selected $User$ record belongs to the region set by $sunflower\_home$. The $Movie$ and $Review$ records are selected as usual. The results for this scenario can be found in Figure 5. These are generally as expected, with SLOG and Detock showing a slight decrease in throughput and an increase in latency as the $sunflower\_chance$ gets higher. As Calvin and Janus do not use regions, they are unaffected. While the sunflower scenario does impact performance for SLOG and Detock, this effect is not as strong as with the baseline scenario. This could be because this scenario still uses the default mh_chance of 50%. This means that even within a 100% $sunflower\_chance$ scenario, half of the $Movie$ and $Review$ records will still belong to the other region. Using a lower mh_chance could put more strain on the sunflower region. Another option would be to use more than two regions. This would allow the other regions to take turns participating in the multi-home transactions, increasing the relative difference in load between the sunflower region and the others.

4.6 Scalability

In the scalability scenario we test the effect of increasing the number of clients submitting transactions to the system at once. The results for this scenario can be seen in Figure 6. These results show that an increase in the number of clients can allow the database systems to better utilize their capacity. We saw the best performance around 10^3 clients. While increasing the number of clients further showed a slight increase in throughput for SLOG, Detock and Calvin, this came paired with huge latency. Although Janus performed poorly in all scenarios, it performed especially badly in this one. The throughput plummeted at 10^4 clients and runs with any higher number failed to finish. Since the other scenarios use the default number of 3000 clients, this could be partially responsible for the low throughput and high P95/P99 Janus suffered from in the other scenarios.

4.7 Network Delay

In the network delay scenario the network has increased latency, slowing down communication between nodes. This de-

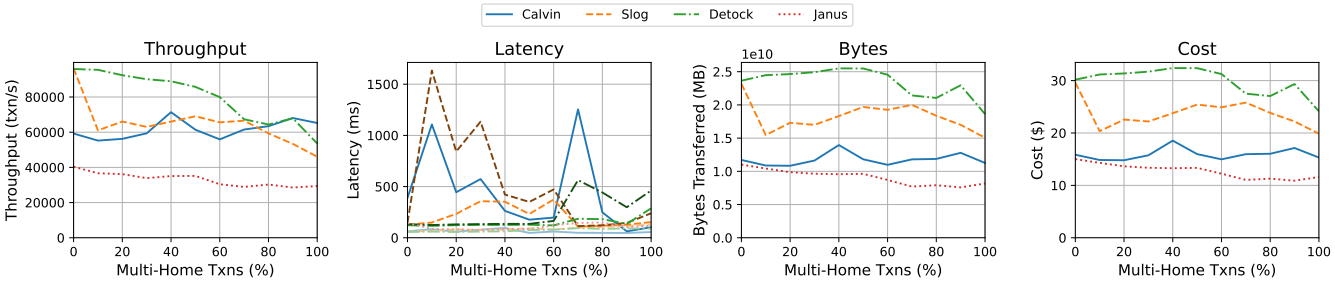

Figure 3: Results baseline scenario

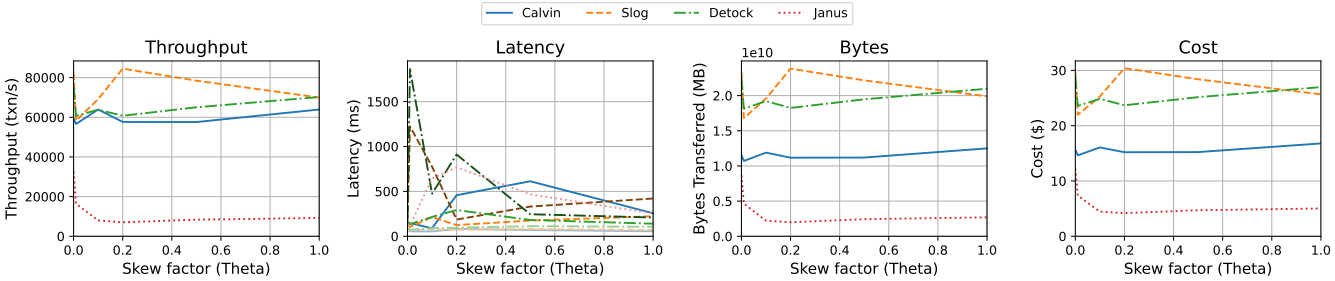

Figure 4: Results skew scenario

lay was simulated using the *netem* [15] Linux tool. For every ms of added latency, we also added 0.1 ms of jitter. The results can be found in Figure 7. As expected, the throughput steadily decreases for all systems as the latency increases. However, we find that while Detock generally performs well in the other experiments, it performs quite badly here. Any delay from 100ms onward resulted in tiny throughput and huge latency. This is despite Detock being specifically designed for processing multi-region transactions with minimal latency.

4.8 Packet Loss

In the packet loss scenario, a certain percentage of the packages is dropped, testing the performance on an unreliable network. This was also simulated using *netem*. The results for the packet loss scenario can be found in Figure 8. These results are again as expected, with throughput decreasing and latency increasing when more packets are lost. We see the same pattern as with the network delay scenario, with Detock proving to be particularly sensitive to network disruptions. However, the difference with the other systems is not quite as drastic in this scenario.

5 Responsible Research

To improve reproducibility, the repository for this project will include a *README* file describing the steps needed to run the experiment. It will also include example commands where needed. The code for this project will include code comments to improve readability and encourage extension.

5.1 Benchmark environment

While an effort was made to keep the benchmarking environment consistent, it could be hard for other researchers to replicate it and obtain the same results. The benchmark was run on a TU Delft cluster, which uses expensive server-grade hardware and is not accessible to outsiders. Since this cluster is used for other research projects as well, a spiking load from these projects could have affected the experiment and made the results unpredictable.

A solution to this would be to rerun the experiment in a more easily reproducible environment with minimal variability. This could be achieved with either dedicated, commodity hardware (e.g., a Raspberry Pi cluster) or a cluster hosted on a commercial cloud provider like AWS.

6 Discussion

This section discusses the limitations of our research and puts our results in a broader context. Section 6.1 starts by describing the limited combinations of parameters we used. Section 6.2 follows by explaining the inherent subjectivity in adapting DeathStar Movie. Section 6.3 compares our results to those from other research. Section 6.4 ends by listing our recommendations for future research.

6.1 Combining parameters

A major limitation of this paper is the fact that all of our scenarios only changed one parameter at a time. For example, other work includes experiments which change both the mh_chance and skew factor [4]. This also makes the benchmarks more realistic, as a real database deployment would be dealing with a mixture of all the scenarios we presented. In particular, it would be valuable to combine the network delay

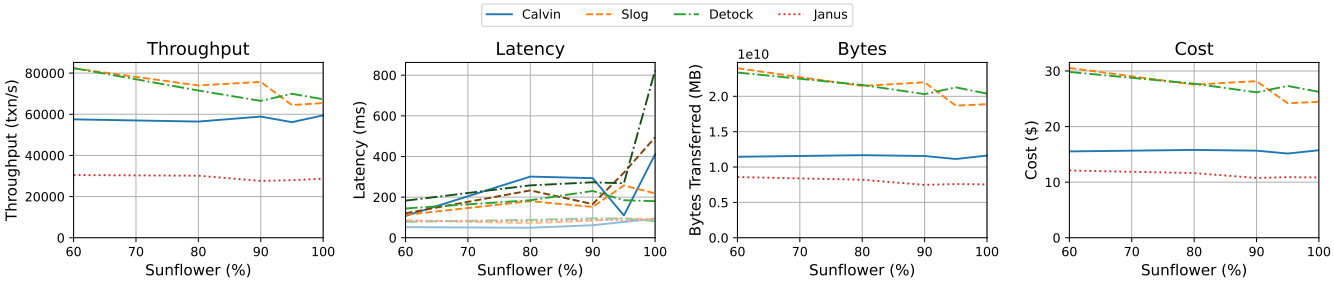

Figure 5: Results sunflower scenario

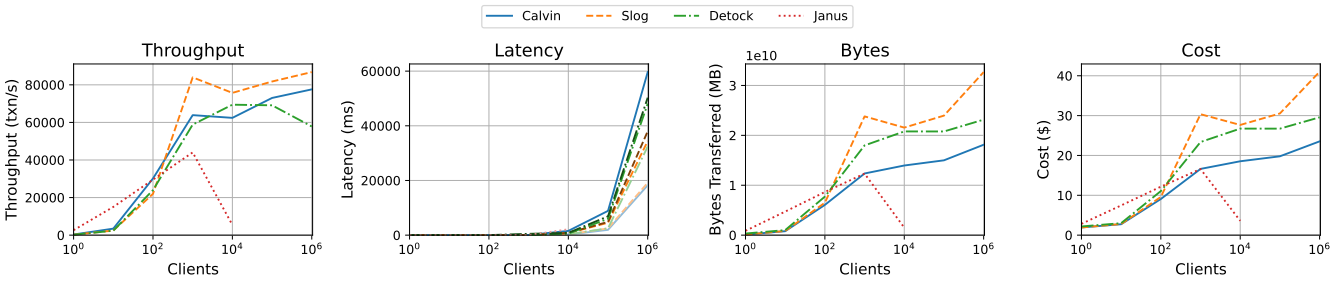

Figure 6: Results scalability scenario

scenario with the others, as a geo-distributed deployment will always have some significant inter-region latency. As mentioned in Section 4.5, combining the sunflower scenario with the baseline scenario could make its effect more visible.

Another possibility for improving sunflower involved the use of more than two regions. This would be more realistic overall, as an application operating at a global scale would likely use many regions. The amount of machines hosting the database could also be used as an additional parameter. This could then be combined with the scalability scenario to test the horizontal scalability of the systems.

Future research could further investigate these combined scenarios, though it would most likely only be able to include a subset of the parameters seen in this paper.

6.2 Interpretation of DeathStar Movie

This paper uses a modified version of DeathStar Movie to highlight its use for geo-distributed databases and to make it more comparable to industry-standard benchmarks like TPC-C and YCSB-T. However, this version is only one of many possible interpretations of how to convert DeathStar Movie to a benchmark for geo-distributed databases. Another research team evaluating DeathStar Movie could come up with a different interpretation, like keeping more redundancy or keeping the microservices. This would almost certainly lead to different results.

Since the adaptation of DeathStar Movie to a benchmark for geo-distributed databases is such a subjective process, there is no straightforward solution to this problem. One option would be to give a new name to the version of DeathStar Movie used in this paper to distinguish it from the original.

6.3 Related work

While there are no other papers evaluating DeathStar Movie for benchmarking geo-distributed databases, we can compare our results to those from other benchmarks. The Detock paper [4] evaluates SLOG, Detock, Calvin and Janus using YCSB. The results from those experiments line up well with ours. Particularly, they show Calvin's ability to beat both SLOG and Detock when handling high multi-home% transactions, even though its design is more primitive. It also shows Janus performing the worst, especially when data access is heavily skewed.

6.4 Recommendations for future work

As mentioned above, we would like to see research combining multiple of the scenarios that appear in this paper.

In addition, we believe it would be valuable to conduct a deeper investigation into the root cause of some of the more surprising results from our experiment. Janus performed poorly overall, while Detock fell far behind in network-limited scenarios. Understanding the cause of these problems could help with designing robust geo-distributed systems in the future.

Finally, we recommend expanding the benchmark further. While our version of DeathStar Movie is faithful to the original by including only one transaction in the workload, this makes it rather simple. Implementing multiple transactions, like TPC-C and YCSB-T, could help it to become more realistic and well-rounded. These transaction could make use of some of the currently unused MongoDB databases from the original DeathStar Movie.

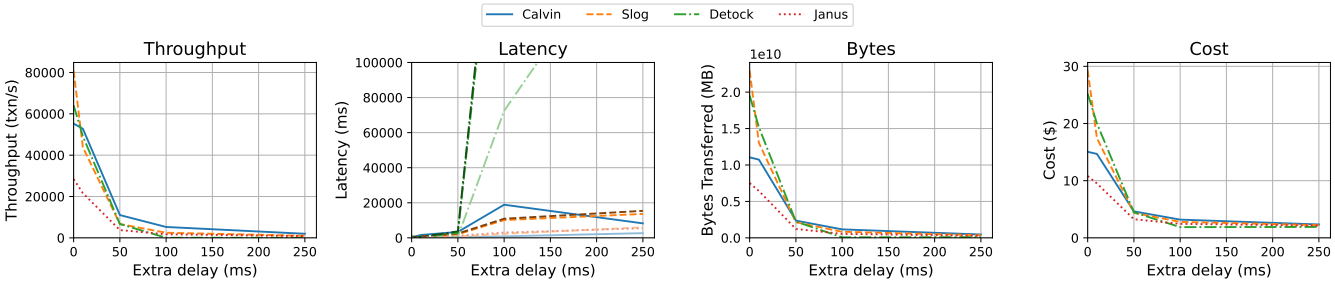

Figure 7: Results network delay scenario

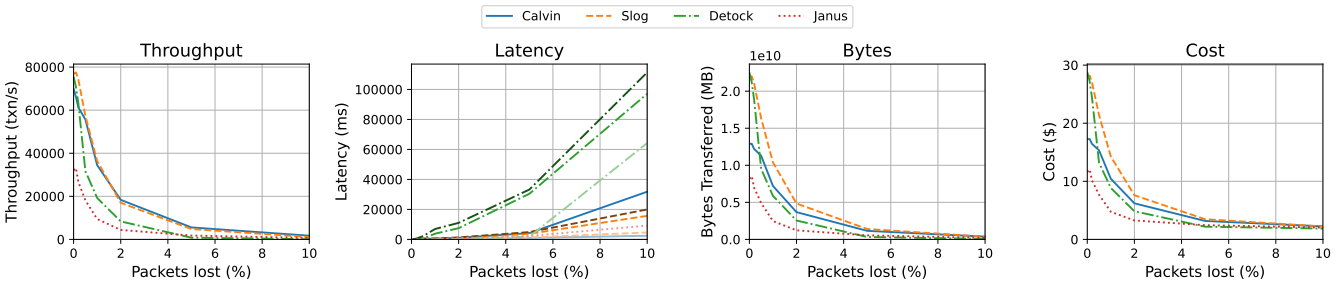

Figure 8: Results packet loss scenario

7 Conclusion

We designed an adapted version of the DeathStar Movie benchmark for testing geo-distributed databases. We then conducted an experiment that tested four database systems and six benchmark scenarios. This showed our adaptation of DeathStar Movie to be a capable benchmarking tool, exposing the strengths and weaknesses of the various systems it was tested with. Janus had bad performance overall, with a particular weakness in scenarios with high contention. We also observed poor performance from Detock in scenarios constraining the network. This is notable due to its advertised low latency for cross-region transactions. To explore the capabilities of our implementation even further, we would like to see larger-scale tests, along with tests combining multiple scenarios into one.

References

[1] Jeff Shute, Mircea Oancea, Stephan Ellner, Ben Handy, Eric Rollins, Bart Samwel, Radek Vingralek, Chad Whipkey, Xin Chen, Beat Jegerlehner, Kyle Littlefield, and Phoenix Tong. F1 - the fault-tolerant distributed rdbms supporting google's ad business. In *SIGMOD*, 2012. Talk given at SIGMOD 2012.

[2] Tim Liu, Mundhra Rohit, and Alex Bai. Talk presented at the HTAP Summit, September 2024. Available at https://www.youtube.com/watch?v=Ddk6k9Js0nI.

[3] Kaustubh Beedkar, David Brekardin, Jorge-Arnulfo Quiané-Ruiz, and V. Markl. Compliant geo-distributed data processing in action. *Proc. VLDB Endow.*, 14:2843–2846, 2021.

[4] Cuong DT Nguyen, Johann K. Miller, and Daniel J. Abadi. Detock: High performance multi-region transactions at scale. *Proceedings of the ACM on Management of Data*, 1(2):1–27, 2023.

[5] Transaction Processing Performance Council. TPC Benchmark™ C Standard Specification, Revision 5.11, 2010.

[6] Akon Dey, Alan Fekete, Raghunath Nambiar, and Uwe Röhm. Ycsb+t: Benchmarking web-scale transactional databases. In *2014 IEEE 30th International Conference on Data Engineering Workshops*, pages 223–230, 2014.

[7] Luyi Qu, Qingshuai Wang, Ting Chen, Keqiang Li, Rong Zhang, Xuan Zhou, Quanqing Xu, Zhifeng Yang, Chuanhui Yang, Weining Qian, and Aoying Zhou. Are current benchmarks adequate to evaluate distributed transactional databases? *BenchCouncil Transactions on Benchmarks, Standards and Evaluations*, 2(1):100031, 2022.

[8] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 3–18, New York, NY, USA, 2019. Association for Computing Machinery.

[9] Christina Delimitrou. Deathstarbench movie review application. https://github.com/delimitrou/DeathStarBench/tree/master/mediaMicroservices, 2025. Accessed: 2025-06-02.

[10] Rachael Harding, Dana Van Aken, Andrew Pavlo, and Michael Stonebraker. An evaluation of distributed concurrency control. *Proc. VLDB Endow.*, 10(5):553–564, January 2017.

[11] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 1–12. ACM, 2012.

[12] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. Consolidating concurrency control and consensus for commits under conflicts. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*, pages 517–532. USENIX Association, 2016.

[13] Kun Ren, Dennis Li, and Daniel J. Abadi. Slog: Serializable, low-latency, geo-replicated transactions. *Proceedings of the VLDB Endowment*, 12(11), 2019.

[14] Amazon ec2 on-demand pricing. https://aws.amazon.com/ec2/pricing/on-demand. Accessed: 2025-06-16.

[15] Stephen Hemminger. Network emulation with netem. *Linux Conf Au*, 05 2005.