Benchmarking geo-distributed databases

Evaluation using the DeathStar hotel reservation benchmark

Aidan Eickhoff[1]

Supervisors: Asterios Katsifodimos, Oto Mráz

[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 18, 2025

Name of the student: Aidan Eickhoff
Final project course: CSE3000 Research Project
Thesis committee: Asterios Katsifodimos, Oto Mráz, Koen Langendoen

Abstract

As modern applications become more global and resource intensive, geo-distributed databases have become critical for fast, reliable data storage. Evaluating the performance of these databases through traditional benchmarks such as TPC-C and YCSB-T is not sufficient to expose all characteristics of the database's performance. A deeper analysis of available benchmarks is needed to determine how to effectively asses geo-distributed databases.

In this paper, we present a modified implementation of the DeathStar hotel benchmark fit for relational databases and use it to evaluate several geo-distributed databases, Detock, SLOG, Janus, and Calvin. We then analyze in detail the unique characteristics of the benchmark and the performance of these databases in the context of the benchmark.

1 Introduction

As modern applications require fast, reliable, and scalable databases, geo-distributed database systems have emerged as a critical solution to the problem. By strategically distributing data across multiple regions, users are closer to databases, improving latency and performance while also increasing redundancy and fault tolerance[1]. Additionally, the inherently scalable nature of these systems can meet the requirements of ever growing modern applications[1]. As a result, geo-distributed databases are frequently used for modern, large-scale, global applications.

Geo-distributed database system performance is a critical metric when deciding which system to use for a given application. There are many methods of testing or reasoning about the performance of a database system, but the best is usually a benchmark[2]. A benchmark will use a set database configuration and generate a set of actions (transactions) for the database to perform and measure the resulting performance[2], which allows repeatable simulation of a real workload. Throughout the past several standard benchmarks have been developed, most notably TPC-C[3] and YCSB-T[4]. Although these benchmarks (and several others) have worked reasonably well, they were not designed for geo-distributed databases and do not accurately reflect modern workloads[5], most notably lacking any form of storage movement.

As a result, there is potential for improved benchmarks that better reflect modern workloads to be adopted. While this potential is investigated, there is a need for a comprehensive comparison between common benchmarks. While benchmarks have been evaluated for suitability in surveys[5], a more accurate comparison requires benchmarks to be implemented using a common framework, which has not yet been done. An accurate comparison would allow researchers to identify flaws in each benchmark and potentially propose new ones which could address these flaws. One of many such benchmarks requiring implementation and evaluatation is the DeathStar[6] hotel reservation benchmark, which emulates a modern travel booking application. The aim of this project is to implement the DeathStar hotel reservation benchmark and use it to evaluate several geo-distributed databases. Additionally, four other group members will implement their own benchmarks, enabling an accurate comparison of all five benchmarks as well as industry standard ones.

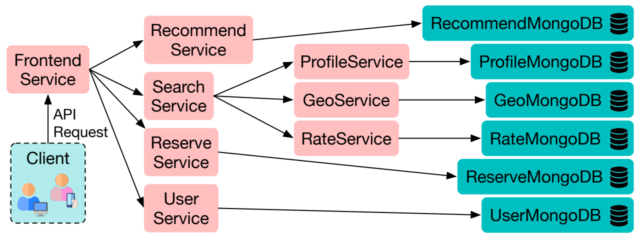

Figure 1: Original benchmark schema from [11]

1.1 Research Question

How do geo-distributed database systems perform when benchmarked using the DeathStarBench hotel reservation benchmark?

We will measure performance using latency, throughput, byte transfers, and cost. This evaluation will take place under different conditions which mimic real-life scenarios, which will be discussed in more detail in Section 3.

1.2 Paper overview

In this paper, we present an implementation of the DeathStar hotel benchmark within the Detock framework (Section 3) and evaluate several geo-distributed databases using this benchmark (Section 4), namely Detock[7], Calvin[8], SLOG[9], and Janus[10]. Our analysis (Section 5) reveals new insights about the performance characteristics of each database and the suitability of the DeathStar hotel benchmark as a geo-distributed database benchmark. Finally, we discuss the ethical considerations and reproducibility of the research (Section 6) and give our conclusions (Section 7).

2 Background

In this section, we will begin by providing a brief overview of the DeathStar hotel benchmark in Section 2.1, followed by a discussion of the databases under test in Section 2.2

2.1 DeathStar

The DeathStar suite of benchmarks was designed to benchmark datacenter hardware and networking performance. The hotels benchmark is meant to simulate a modern travel booking application which allows users to search for, be recommended, and reserve hotels. It is designed using microservices and tools such as nginx and memcached to mimic the behavior of a modern cloud-based application. A diagram of the path requests follow can be seen in Figure 1. When a transaction reaches the frontend service, it is forwarded to its respective service, which handles the execution. As each microservice is solely responsible for one transaction, it has a mongoDB instance containing the necessary data.

The hotel benchmark uses a workload composed of the following 4 transactions:

- User login with username and password.
- Search for an available hotel by date and location.
- Recommendation of a hotel based on location, price, or rating.
- Reservation of a hotel for a given set of dates.

In the open-source implementation of the workload[1], the transaction mix is 0.5% user logins, 60% searches, 39% recommendations, an 0.5% reservations. Because the first three transactions are read-only, this workload is very read heavy.

2.2 Databases under test

The following databases represent state-of-the-art geo-distributed systems, all available in the same repository and able to be benchmarked using a consistent architecture. Each system offers distinct advantages and disadvantages, showcasing some of the latest advancements in distributed database technology. Below, we provide a brief overview of each evaluated system.

Janus

Janus[10] uses a global Paxos process to schedule all transactions. This means all non-conflicting transactions can be scheduled in a single round trip of the network, and all conflicting transactions can be scheduled in two round trips of the network in terms of latency. This was an improvement over previous geo-distributed databases at the time of Janus's development, however better techniques have emerged over recent years.

Calvin

Calvin[8] also uses a global scheduler for every transaction, although one node of the database is designated as the scheduler rather than a Paxos process. Consequently, throughput is very consistent when varying the type and complexity of transactions. This makes Calvin's relative performance very dependent on the workload when compared to other modern geo-distributed databases. Because the ordering process is handled by a single node, latency is also highly dependent on the start location of a transaction.

SLOG

SLOG[9] is a modern geo-distributed database built upon Calvin which uses the concept of regions to increase the throughput of the database. Regions are "a set of servers that are connected via a low-latency network"[9, p. 3]. Transactions are categorized based on whether all the data accessed is in the same region. If it is, the transaction is single-home and can be forwarded to the home region for processing. SLOG takes advantage of this by only using a global ordering process for multi-home transactions.

Detock

Detock[7] uses the same notion of regions as SLOG and processes single-home transactions in a similar fashion, however instead of a global ordering process for multi-home transactions uses a deterministic scheduling algorithm to allow each node to independently arrive to the same schedule. The deterministic scheduling algorithm can significantly reduce latency and increase throughput when compared to a global consensus algorithm as less cross-region coordination is required.

[1]https://github.com/delimitrou/DeathStarBench/tree/master/hotelReservation

3 Implementation

In this section we begin with a brief motivation of the implementation framework in Section 3.1. Design choices made in the adaptation process are outlined in Section 3.2, followed by more details on transaction implementation in Section 3.3. Finally, Section 3.4 outlines the scenarios used to test the databases.

3.1 Framework

Development of the benchmark was done using the the Detock codebase. The Detock codebase contains several implementations of other common geo-distributed databases and a framework for running benchmarks on these geo-distributed databases. These two qualities allow for accurate comparison both between geo-distributed databases and between benchmarks. The ability for accurate comparison between benchmarks is the most important quality within the context of this project and is a large motivation for using the Detock codebase to implement and run the benchmark.

3.2 Adaptation of the benchmark

The adaptation process which is required to use the benchmark as a database benchmark can be done in several manners. In a very literal adaptation, each microservice would be given its own instance of the database under test(replacing mongoDB), endpoints could be replicated in a similar fashion to database partitioning and replication, and the benchmark could be run as normal. Another approach would be to translate the separate databases from each microservice to a single relational database and adapt the requests from the original benchmark into transactions for the new benchmark.

While the first approach more accurately resembles the original benchmark, it has several flaws which make it impractical for this experiment. Firstly, there are scenarios where the database performance is not the limiting factor which makes comparison between databases in these scenarios challenging. Secondly, the first approach would make it difficult to accurately compare to the other benchmarks which are being implemented in parallel to this one as it would not be in the same style. Finally, the first approach adds additional complexity and overhead by requiring coordination and communication between multiple database instances. As a result, we will use a modified version of the benchmark which translates the original queries to transactions and the original set of databases to a relational schema (Figure 2).

The adaptation process was not straightforward as the microservice architecture of the original benchmark meant lots of data duplication occurred. As such, some databases from Figure 1 were combined or modified. Specifically the RecommendMongoDB, the ProfileMongoDB, the RateMongoDB, the GeoMongoDB, and the capacity collection from the ReserveMongoDB were combined to the make the hotels table.

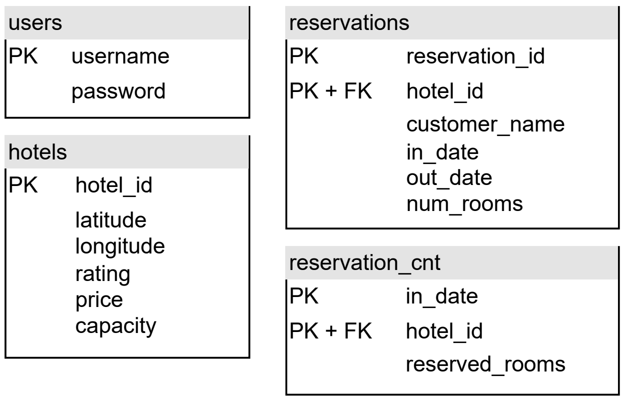

Figure 2: Adapted database schema

The UserMongoDB and ReserveMongoDB were both translated one-to-one into the user and reservations tables respectively. Finally, the adaptation required the addition of the reservation_cnt table to avoid querying all records from the reservations table during the reservation transaction.

3.3 Transaction implementation

Due to the adaptation of the original queries to stricter transactions, there were also some modifications made to the behavior of the transactions. The search and recommendation transactions would originally query all hotel records to find the optimal result. This has the consequence of not allowing single home or single partition queries. Because Detock only allows for queries on primary keys and the added complexity of partitioning and homing based on special keys such as latitude/longitude pairs, we decided each transaction would search through a tuneable k hotels to find the appropriate results. This also allows for skewed data access in all transactions which is important for the skewed testing scenario.

As mentioned before, the reservation transaction must check if a hotel is fully booked during the desired dates. The original implementation would query all previously made reservations and calculate the capacity on dates which overlapped with the desired dates. While this may be appropriate for a datacenter benchmark where heavy computation is expected, this is not efficient compared to adding another table which stores the information and updates it upon each completed reservation, and transactions should not be accessing the entire database for the purpose of a database benchmark. The user login transaction was left unchanged. These modifications create a more interesting and realistic workload for database benchmarking than the original queries of the DeathStar hotel benchmark.

For the search and reservation transaction, a random date range is needed to book the hotel. This was generated by choosing a random date within a range, and choosing a random length d for the range to be.

3.4 Scenarios

The databases were tested under different conditions, each emulating different real-world scenarios. These scenarios are the following:

1. **Baseline scenario**: Increasing number of multi-home transactions.
2. **Skewed scenario**: Increasing access frequency for specific items.
3. **Sunflower scenario**: Transaction region follows distribution based on elapsed time.
4. **Network latency scenario**: Increasing network latency.
5. **Packet loss scenario**: Increasing chances of packet loss.
6. **Scalability scenario**: Increasing number of clients making requests.

For the first three scenarios, it is relevant to discuss the record sampling process in more detail. When creating a transaction, the system randomly determines if it will be single-home or multi-home and whether it will be single-partition or multi-partition. Based on these decisions, the following can be said about the read-write set of the transaction:

- **Single-home**: the read-write set will contain records from the local region (except during the sunflower scenario).
- **Multi-home**: the read-write set will contain either a single record from a foreign region or records from at least two regions, with each records' home being randomly chosen.
- **Single-partition**: the read-write set will contain records from a single randomly chosen partition.
- **Multi-partition**: the read-write set will contain records from at least two partitions, with each records' partition being randomly chosen.

This process is not as straightforward when evaluating the databases under the sunflower scenario. When selecting records for a multi-home transaction each record's home is chosen according to the sunflower distribution. Additionally, single-home transactions in the sunflower scenario access regions according to the region distribution rather than the local region.

In the skewed scenario, each time a record is chosen the system randomly decides if the record will come from the hot set or the cold set. The size of the hot record set is decided beforehand but is dynamically increased for transactions which require more records than the size of the hot set. This ensures all transactions can access their required number of records without becoming undersampled.

4 Results

In this section we first document the experimental parameters in Section 4.1. Then, in Sections 4.2, 4.3, 4.4, 4.5, 4.6, and 4.7 we present the results of the baseline, skew, sunflower, network latency, packet loss, and scalability scenarios respectively. We then present a latency breakdown for each database in Section 4.8. Finally, we have small variations on the baseline scenario and latency breakdown for a smaller value of k in Section 4.9.

4.1 Experimental parameters

Experiments were run on the TU Delft compute cluster using four machines. Each machine used a dual-socket AMD

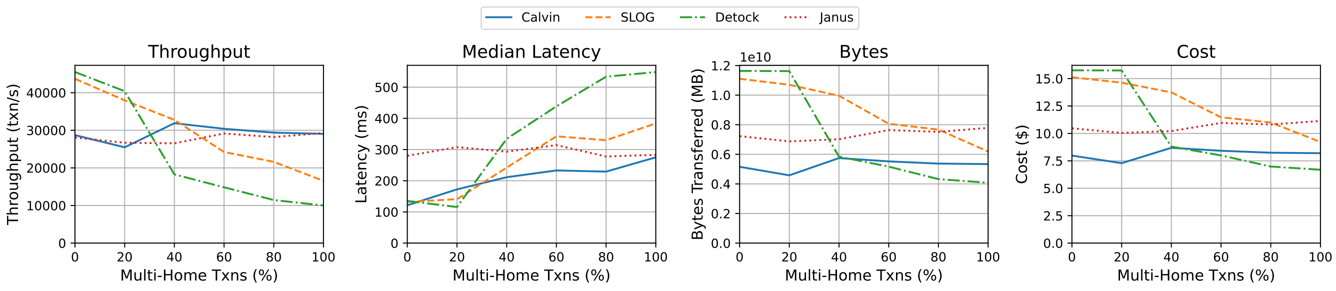

Figure 3: Baseline scenario performance

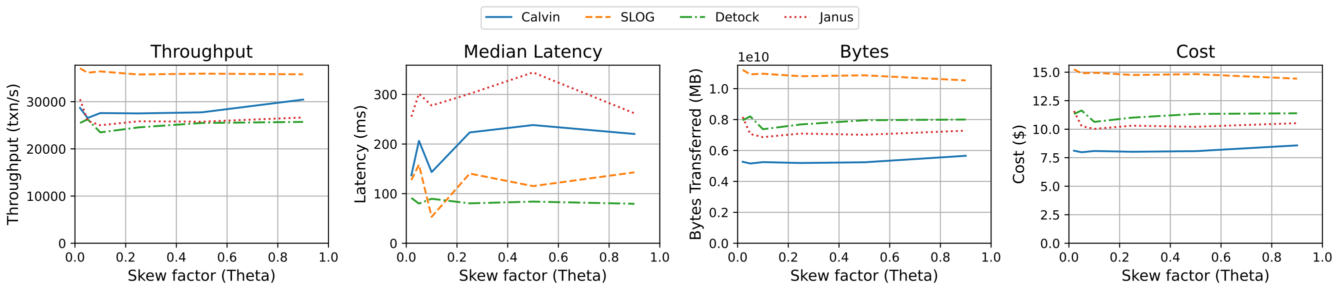

Figure 4: Skewed scenario performance

EPYC 7H12 CPUs with 64 cores per CPU (for a total of 128 physical cores and 256 logical processors per machine) and approximately 515 GB of ram. Latency between machines can be seen in Table 1, as calculated using ping with 100 packets and an interval of 200ms between each packet. All experiments had 2 partitions and 2 regions. Machines st1 and st2 comprised one region, while st3 and st5 comprised the other, each machine responsible for one partition. Requests were generated for region 1 by st5, and for region 2 by st2.

Table 1: Network Latency Matrix (Average RTT in milliseconds)

Source / Dest	st1	st2	st3	st5
st1	–	0.125	0.131	0.084
st2	0.118	–	0.124	0.073
st3	0.137	0.137	–	0.073
st5	0.089	0.095	0.064	–

Unless otherwise specified, experiments were run with 25% chance of a transaction being multi-partition, a 25% chance of a transaction being multi-home, and with 3000 clients generating requests. In the database configuration, there were 500 users and 200 hotels and dates were generated from a range of 6 months with a maximum stay length of $d = 14$. Search and recommendation transactions accessed $k = 10$ hotels each.

Data is collected in the form of throughput in transactions per second, median latency, total bytes transferred, and the simulated cost of running the benchmark on AWS (factoring in rental costs and bytes transferred).

4.2 Baseline scenario

The results for the baseline scenario can be seen in Figure 3. The percent of transactions which are multi-home is varied from 0% to 100%. The user transaction becomes a foreign single-home transaction because it only touches a single record.

From figure 3, both Janus and Calvin have relatively constant throughput as the amount of multi-home transactions increases, as these databases use a global ordering procedure to avoid conflicts between transactions. Detock and SLOG both avoid this global ordering process for single-home transactions, and as a result have high throughput when the amount of multi-home transactions is low, though this drops significantly as more multi-home transactions are introduced. Bytes transferred and cost follow similar trends to throughput for each individual database, though it should be noted Calvin and to a lesser extend Janus are much more efficient than Detock and SLOG for these metrics.

4.3 Skewed scenario

The chance of a record being taken from the hot record set was 90% during this experiment. The independent variable was the size of the hot record. The number of hot records per partition and region was calculated using the skew factor θ times the total number of records for that partition and region. This means as θ approaches the chance of a hot record being selected, the behavior should be same as the baseline scenario. As a result, the graph goes from high skew where the hot record set is 1 record per region-partition at $\theta = 0.025$ to an unskewed scenario at $\theta = 0.9$.

The size of the hot record set did not significantly impact the performance of any of the databases.

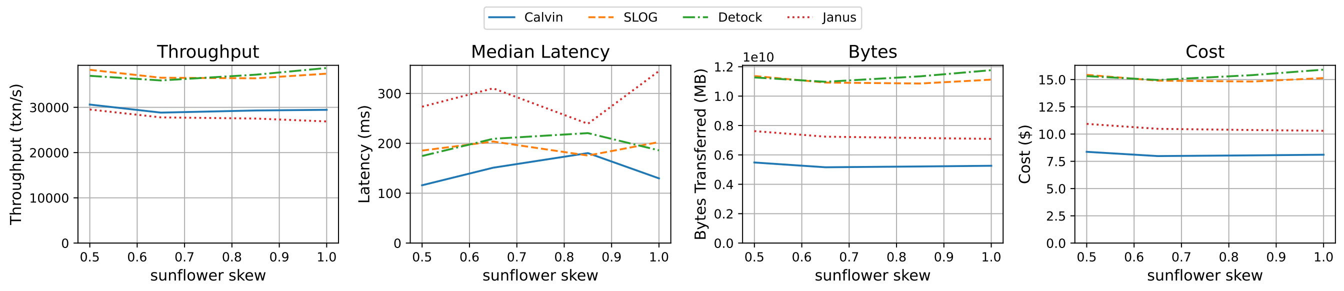

Figure 5: Performance with sunflower skews

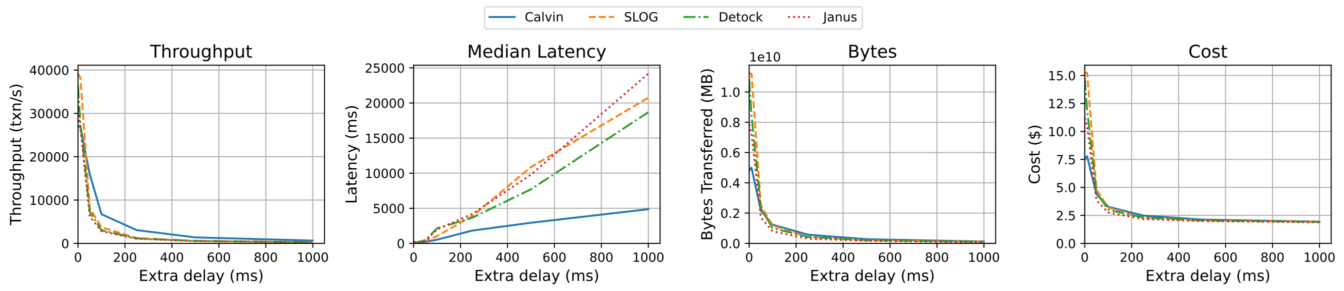

Figure 6: Performance with increasing latency

4.4 Sunflower scenario

Table 2: Sunflower scenario transaction region weights

Elapsed txn count	Region one	Region two
0-20%	.5	.5
20-40%	x	$1 - x$
40-60%	.5	.5
60-80%	$1 - x$	x
80-100%	.5	.5

This experiment changes the home region of single-home transactions based on the elapsed of the experiment. In this implementation, transactions' home region followed a distribution based on the total number of sent transactions. While ideally the home region would be based exactly on the elapsed time, transactions are pre-generated which makes this infeasible. As a result, elapsed time is approximated using the number of sent transactions. In this experiment, the transactions generated are homed using a distribution according to Table 2, where x is the sunflower skew.

From Figure 5, it is clear the performance is not significantly impacted by skewing the home region of transactions.

4.5 Network latency

This experiment artificially increased latency between all machines on the network. This means the latency is felt both between regions and between partitions in a region, which should normally be very low latency. The results are visible in Figure 6.

As a result of the increased latency between nodes, the throughput of all databases drops steeply and latency quickly increases beyond reasonable levels. Steep drops in bytes transferred and cost as latency increases are also present. Even Detock which does not rely on a global consensus algorithm for transaction ordering, must communicate all transactions to involved regions which causes the drop in throughput. Calvin suffers the least from the increased latency, likely because it requires the least cross-region/cross-partition communication (in terms of raw bytes transferred).

4.6 Packet loss

This experiment artificially stops some packets sent between machines. For the same reasons as the increased latency experiment, the coordination between machines causes a steep decrease in throughput and increase in latency which can be seeing in Figure 7.

All databases suffer large throughput drops and latency increases as packet loss increases. Bytes transferred and cost decrease similarly to throughput. Similarly to the network latency scenario, Calvin performs the best as packet loss increases, likely for the same reason.

4.7 Scalability

This experiment increases the number of clients making requests to the database system. The reuslts are visible in Figure 8. Throughput increases when there are more clients making transactions until a point where the database gets overloaded. At this point the throughput remains relatively consistent, though the latency increases significantly. Even though the throughput remains relatively consistent, the bytes transferred and consequently the cost increased as the number of clients increased.

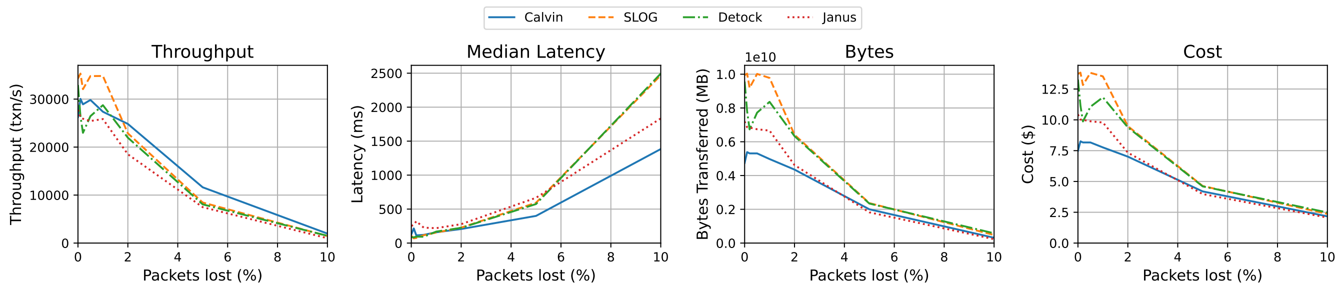

Figure 7: Performance with increasing packet loss

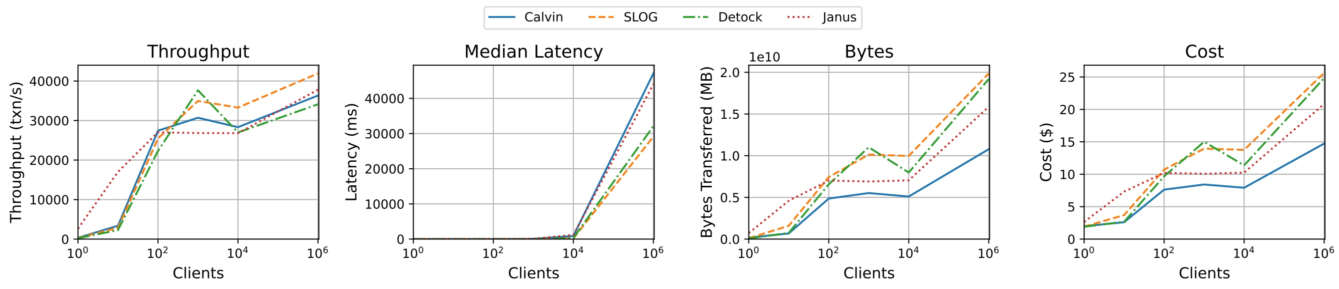

Figure 8: Performance with scaling client numbers

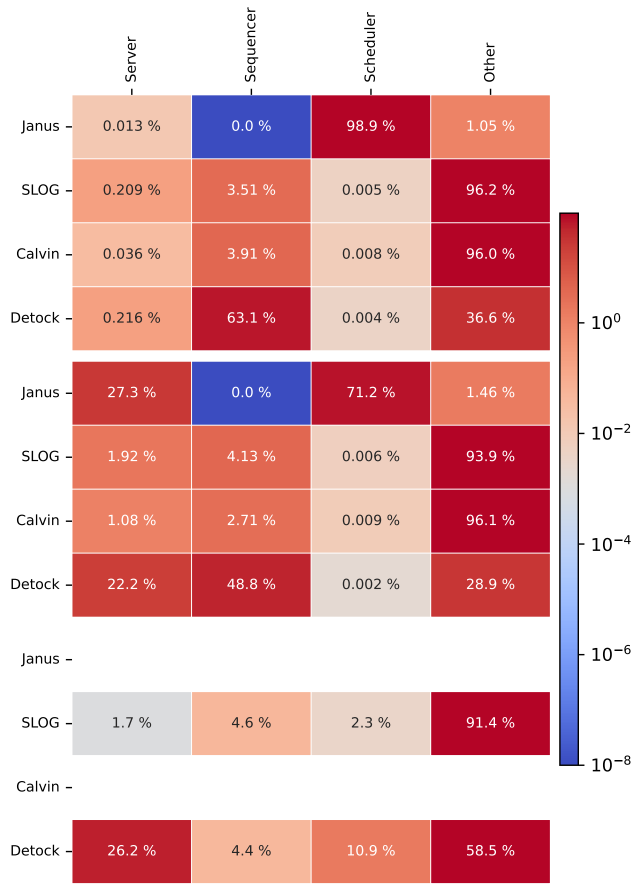

Figure 9: Latency breakdown

4.8 Latency breakdown

This experiment provides more detailed breakdown of how latencies are comprised for each database in the baseline scenario. The four main latency categories are:

- **Server**: Responsible for transaction forwarding and operations (reading/writing).
- **Sequencer**: Handles multi-home ordering and log management.
- **Scheduler**: Handles transaction scheduling (including deadlock resolution in Detock's case) and lock management.
- **Other**: All remaining latency sources.

The exact details of what happens in each latency component varies slightly for each database, and is beyond the scope of this paper.

The first four rows of Figure 9 show the latency components of a single-home, single-partition transaction for each database. The second four rows show the latency components of a single-home, multi-partition transaction. With the exception of Janus which waits nearly exclusively on the scheduler, and Detock which spends some time on the sequencer, the majority of the latency is coming from the Other category. The last two rows show the latency components of a multi-home transaction, however since Calvin and Janus do not include the idea of homes they are not included. Here, again, the majority of the latency is in the Other category.

4.9 Transaction variations

When evaluating the results of the baseline and the latency breakdown, we were curious about the cause of differences in these results and typical benchmark results (e.g. TPC-C or

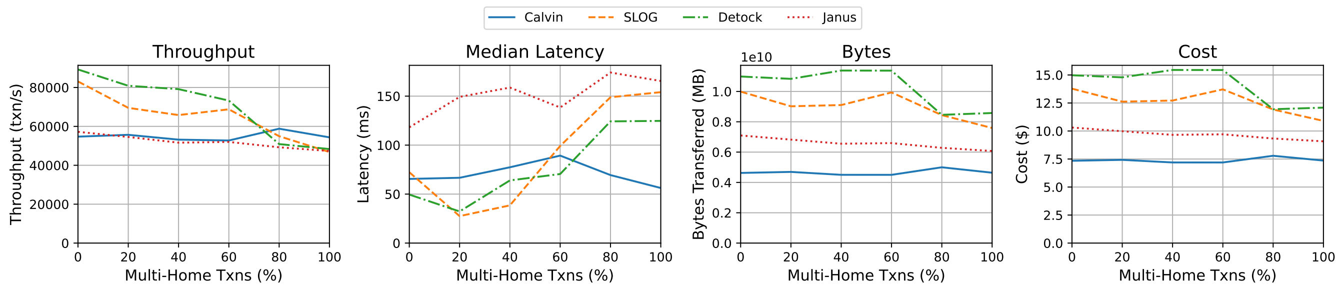

Figure 10: Baseline scenario with smaller transactions

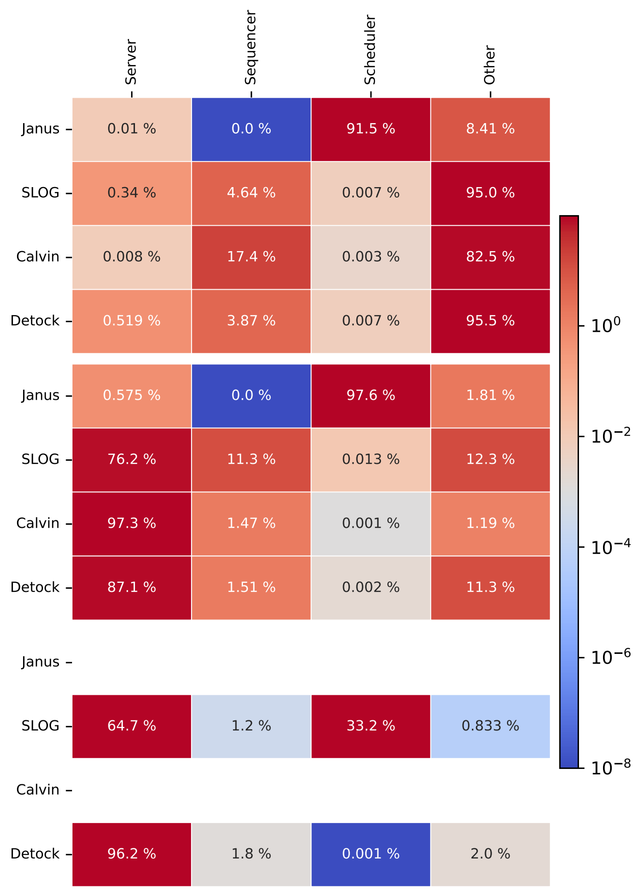

Figure 11: Latency breakdown with smaller transactions

the other benchmarks being implemented in parallel). As a result, we provide a small test of these scenarios with modified parameters, namely a smaller read size of $k = 3$. This should be a better test of the databases' coordination abilities as less time will be spent reading each transaction, and there will be considerably more transactions.

The results of the baseline scenario are shown in Figure 10. Compared to the results of the baseline scenario with $k = 10$ (Figure 3), we can see similar trends: Janus and Calvin remain relatively constant, while SLOG an Detock's throughput decline as more multi-home transactions are present. However, SLOG and Detock perform notably better under these conditions, as their decline in performance is far less severe. The median latencies for all databases are also significantly lower, though this is expected. Despite Calvin outperforming the other databases in terms of throughput and latency, it also has the lowest bytes transferred and lowest cost, highlighting it's efficient communication.

The latency breakdown reveals additional information about the difference between these conditions. Visible in Figure 11, the single-home single-partition latency components are very similar to those from the $k = 10$ breakdown (Figure 9), with the exception of Detock which spends less time on the Sequencer. This is not the case for multi-partition and multi-home transactions where the majority of the latency is now caused by the Server. This points to increased cross-region and cross-partition communication requirements as more throughput occurs.

5 Discussion

This section will begin by mentioning some unusual qualities of the DeathStar hotels benchmark (Section 5.1), focusing on the read and processing intensive transactions. Following, we discuss the results in more depth with potential justification for some of the more unusual results this benchmark presents (Section 5.2).

5.1 Nature of the benchmark

The read heavy nature of the benchmark must be considered when using the DeathStar hotel reservation benchmark. Furthermore, writes occur to a database with relatively many records (number of hotels times number of available dates). These features of the benchmark means most transactions will not conflict with each other and the scheduling efficiency or deadlock avoidance of a system is not thoroughly tested. For some modern workloads, this is not relevant as read performance is critical, though for others this may provide a critical feature of database performance which the DeathStar hotel reservation benchmark misses. As a result, this scenario could only be an appropriate benchmark if the eventual workload is known to be very read heavy and with few write conflicts.

The benchmark also contains transactions which are rather processing intensive. The recommendation query will select at least k (recommendation read size) records, and the search query will select at least $k + d$ records where d is the length of the stay in days (random between 1 and 14) and at most $k * d$ records.

5.2 Database performance

As a result of the read-heavy nature discussion of database performance can only happen in a limited scope. This limitation is clearly visible in the baseline scenario. As discussed in section 2, Janus requires 2 network round trips to resolve conflicting transaction orders. This will rarely happen in the hotel benchmark as conflicting transactions are exceptionally rare. As a result, the performance compared to other databases is significantly better than demonstrated in relevant literature [7; 8]. The advantage of SLOG and Detock is also visible in the baseline scenario when there is a low percentage of multi-home transactions. This advantage is likely less pronounced under the DeathStar hotel benchmark than others due to the processing intensive nature of the transactions. This type of transaction would favor raw read speeds rather than advanced coordination techniques.

This is further evidenced by the results from the baseline scenario with $k = 3$, where Detock and SLOG perform considerably better compared to Janus and Calvin than in the scenario with $k = 10$. Furthermore, when comparing the latency breakdown for $k = 10$ and $k = 3$, considerably more time is spent in the Server and Scheduler for multi-home and multi-partition transactions. As Detock and SLOG have more advanced techniques for managing conflicting transactions, when the majority of the latency is spent forwarding, scheduling, and sequencing transactions it makes sense that they perform better.

The skew scenario presents more slightly unusual results as skew has little effect on performance, however this can also likely be explained by the read-heavy nature of the benchmark and a lack of conflicting transactions. None of the reads will conflict with each other, and higher skew will therefore have little impact on creating more conflicts. The relative performance of databases should not be drawn from this graph as the most significant factor affecting this is the transaction's 25% multi-home and 25% multi-partition chance.

Similar to the skew scenario, the sunflower scenario also has little effect on database performance, likely for similar reasons.

The packet loss and latency scenarios show the advantage of Calvin's more efficient cross-region and cross-partition communication, as network issues do not affect performance as significantly as the other databases. The performance degradation in these experiments aligns with expectations.

The latency breakdown reveals some more information about the parameters of the benchmark. As mentioned before, the majority of the latency for the $k = 10$ condition is in the 'Other' category. This category includes idle time, which when combined with the information from the scalability scenario where databases seem to reach full (or near full throughput) as low as 100 clients, it is possible the databases are already overloaded when the benchmark is run with 3000 clients. The $k = 3$ condition does give more readily interpretable results. Clearly, multi-home and multi-region transactions require more coordination and this is demonstrable with higher latency components in the Server, which includes forwarding transaction information to relevant regions and partitions. As the throughput is considerably higher than in the $k = 10$ condition, the databases can additionally handle more clients making requests. The combination of these factors makes this a distinct possibility, although more investigation is necessary to determine the root cause of the different latency components.

6 Responsible Research

Ensuring responsible research practices is critical to the ensure credibility, validity, and the ethical integrity of this study. Given performance evaluation can influence the adoption or improvement of systems, any methodological flaws or ethical issues may cause misleading conclusions to be drawn, directly harming academia or industry. We will first discuss the reproducibility of the study in Section 6.1 and then discuss the ethics in Section 6.2.

6.1 Reproducibility

Reproducibility of an experiment is very important for the validity of the research. Clear documentation of database system configurations, hardware specifications, experimental parameters, and major design decisions are essential as these can have large impacts on the outcomes of an experiment. The documentation discussed above will allow for reproducibility of the experiment as the required steps are clearly articulated. Additionally, the source code will be available at the delftdata Detock repository[2]. Detailed information about how DeathStar hotels experiments were run and corresponding scripts will also be available in this repository. Using these tools, experiment results should be reproducible.

6.2 Ethics

Within the context of the project, it is important to divulge all choices made which could influence the outcome of the experiments. Also important is avoiding selective result reporting, data manipulation, or other parameter manipulation to achieve desired results. These two factors could cause incorrect or incomplete conclusions to be drawn, meaning database performance is not accurately reflected. As discussed earlier, this could influence the adoption or improvement of some systems, potentially harming academia or industry. Through the documentation detailed in Section 6.1, these two problems should be mitigated.

7 Conclusions and Future Work

This study evaluated the performance of distributed databases under varying workloads, revealing key trade-offs in their design choices. In scenarios with a majority of single-home transactions, Detock and SLOG perform well, while Calvin and Janus outperform others when there are many multi-home transactions due to their global ordering procedures, which give them fairly consistent performance. All databases perform reasonably during high skew and in the sunflower scenario, likely because the read-heavy nature of the benchmark reduces contention. In environments prone to network issues,

[2]https://github.com/delftdata/Detock/tree/DeathStarHotels

Calvin emerges as the optimal choice. Additionally, Calvin's efficient communication leads to low operating costs without sacrificing too much performance.

Future work could expand this analysis in several directions. First, adding more databases to the comparison, such as Mencius[12] or Caerus[13] would broaden the evaluation. Additionally, experiments varying the multi-partition nature of the benchmark could give deeper insight into the behavior of these geo-distributed databases. Finally, a more detailed investigation of the latency breakdown, especially the remaining 'Other' category, could bring additional insights into database bottlenecks or the DeathStar hotel benchmark.

References

[1] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD/PODS '20, page 1493–1509. ACM, May 2020.

[2] Yao-S Bing and Alan R. Hevner. A guide to performance evaluation of database systems. Report, National Bureau of Standards (NBS), Washington, DC 20402, 1984.

[3] Transaction Processing Performance Council. Tpc benchmark c revision 5.11. Technical report, Transaction Processing Performance Council, February 2010.

[4] 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.

[5] 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, March 2022.

[6] 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. ACM, April 2019.

[7] Cuong D. T. 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, June 2023.

[8] 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*, SIGMOD/PODS '12. ACM, May 2012.

[9] Kun Ren, Dennis Li, and Daniel J. Abadi. Slog: serializable, low-latency, geo-replicated transactions. *Proceedings of the VLDB Endowment*, 12(11):1747–1761, July 2019.

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

[11] Ka-Ho Chow, Umesh Deshpande, Sangeetha Seshadri, and Ling Liu. Deeprest: deep resource estimation for interactive microservices. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 181–198. ACM, March 2022.

[12] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Mencius: building efficient replicated state machines for wans. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 369–384, USA, 2008. USENIX Association.

[13] Joshua Hildred, Michael Abebe, and Khuzaima Daudjee. Caerus: Low-latency distributed transactions for geo-replicated systems. *Proceedings of the VLDB Endowment*, 17(3):469–482, November 2023.