Benchmarking geo-distributed databases

Evaluating Performance using the SmallBank benchmark

Cirtog Filip-Andrei [1]

Supervisor(s): Dr. Asterios Katsifodimos [1]**, Oto Mraz** [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: Cirtog Filip-Andrei
Final project course: CSE3000 Research Project
Thesis committee: Dr. Asterios Katsifodimos, Oto Mraz, Koen Langendoen

Abstract

In recent years, applications have started using geo-distributed databases, even though their behavior under different workloads remains complex. Therefore, this project analyses how several databases handle transactional workloads using the SmallBank benchmark. We implement and adapt an already existent benchmark, previously used for non-distribuited databases. Furthermore, we use it to evaluate multiple geo-distributed databases highlighting their strengths and weaknesses throughout multiple scenarios designed to stress different parts of the system. We observe that in most scenarios Detock performs slightly better than SLOG, and both outperform Calvin. This aligns with the fact that Detock builds upon SLOG, which itself improves on Calvin. On the other hand, Janus's performance is significantly behind the others due to its communication overhead. However, while the SmallBank benchmark provides an insightful comparison, providing specific advantages compared to previous benchmarks, its adaptation to geo-distributed databases limits its ability to compare scenarios with higher percentages of multi-home and multi-partition transactions.

1 Introduction

Distributed and geo-distributed databases form the backbone of countless critical applications, from global e-commerce stores (Amazon [1]) to finance and banking institutions (A top 5 US Bank [2]) or Social Media platforms (Facebook [3]). However, they come with great complexity regarding design, especially when it comes to their trade-offs between consistency, scalability, and latency. Because the distance between nodes raises latency and slows communication, they require more complex coordination to maintain consistency. Moreover, uneven user distribution across the database nodes leads to further congestion and latency. As a result, we still lack clarity on which system performs best under specific workloads or scales effectively across the globe.

Researchers have previously developed two workloads for benchmarking: TPC-C [4] and YCSB-T [5]. However, besides their limited scope, the differences between academic and industry benchmarks make them less representative of real-world workloads. For instance, YCSB-T focuses on too simple key-value operations, with half involving access to a single row [6]. Its transactions are simple reads and updates over a flat data model, which does not represent the complex workloads that the industry uses. TPC-C is a more complex workload than YCSB-T, which assumes a centralized warehouse model, less common in the real world. Moreover, it cannot create data hotspots, making it unsuitable for evaluating system performance under high-contention workloads [6].

Although some papers use SmallBank, a transactional workload that simulates banking operations, to investigate the end-to-end performance of non-distributed systems [7], there is a lack of detailed assessments concerning latency, throughput, and byte transfers for geo-distributed databases. Therefore, in this paper, we aim to evaluate *how geo-distributed databases perform under the SmallBank benchmark in terms of throughput, latency, and bytes transferred, as well as their operational cost*. Specifically, it assesses the strengths and weaknesses of existing modern geo-distributed systems such as Calvin [8], SLOG [9], Detock [10], and Janus [11].

This paper makes the following contributions: an implementation of the SmallBank benchmark in the Detock codebase, a series of design modifications we applied to SmallBank to support its transition from a non-distributed to a distributed database architecture workload, and an assessment of the performance of each database across multiple experimental scenarios, each designed to stress different aspects of the system.

Subsequent to this introduction, Section 2 presents the background information, including an overview of the databases we evaluate and a description of the benchmark. In Section 3 we describe how we implemented the benchmark and outline the design choices made to adapt SmallBank for geo-partitioned systems. Next, in Section 4, we define the dataset, the scenarios, and the setup used for the experiments. We discuss the results in Section 5, while in Section 6 we provide some highlights, including SmallBank's advantages and limitations. Section 7 presents the responsible research principles and details our data collection. Lastly, Section 9 highlights our conclusions.

2 Background

This section provides relevant information regarding the properties of the databases, whose performance will be evaluated in the next sections. Moreover, we explain the benchmark used for evaluation and define the composing transactions.

2.1 Systems under evaluation

This paper focuses on evaluating four different modern databases, which aim to minimize the cummunication between regions, and thus improving performance in geo-distribuited settings: Calvin[8], SLOG[9], Detock[10] and Janus[11].

Calvin is a geo-distributed database that supports disk-based storage designed without a single point of failure. Calvin sequences all transactions to ensure serializability and supports two replication modes. In synchronous replication, Calvin uses Paxos to ensure all sequencers in a replica group agree on each transaction batch before execution, providing strong consistency, but with higher latency. In contrast, asynchronous replication designates a master sequencer to immediately process and forward transaction batches to replicas, reducing latency, but complicating failover. Both replication modes preserve Calvin's deterministic execution through consistent transaction ordering.

SLOG, implemented on Calvin, avoids the trade-off between strict serializability, low latency writes, and high transactional throughput for workloads that contain physical region locality in data access. In SLOG, each stored data is

assigned a home region and executes single-home transactions without global coordination, achieving low latency and high throughput. However, SLOG employs a global ordering system for multi-home transactions to prevent deadlocks, adding additional latency which could result in serializability violations, OCC aborts, or deadlock [9].

Detock, an improvement of SLOG, guarantees that each transaction, regardless of its type, only needs a single round trip from the initiating region to the participating region. By replacing the global ordering layer with a graph-based concurrency control protocol, Detock resolves deadlocks without aborting transactions, achieving high throughput for high contention workloads.

Similar to Detock, ***Janus*** implements a reordering technique over a dependency graph. It then executes transactions across all replicas and shards deterministically following the graph order. However, Janus synchronously replicates data to every region, so it needs at least one round-trip to all regions to commit, making it slower than Detock.

2.2 SmallBank benchmark

SmallBank replicates a bank environment, where transactions perform simple operations on customers accounts [12]. As shown in Figure 1, its schema consists of three core tables: `Account` (Name, CustomerID), `Savings` (CustomerID, Balance), and `Checking` (CustomerID, Balance), and of five distinct transaction types. Its functionality can be briefly summarized by the following core methods:

1. **Balance(Name):** Returns the sum of the customer's savings and checking account balances.
2. **DepositChecking(Name, Value):** Deposits the specified value into the customer's checking account. It adds or subtracts the value from the savings balance of the specified customer.
3. **TransactSaving(Name, Value):** Performs a deposit or withdrawal operation on the customer's savings account. It adds or substracts the value for the given customer from the savings balance.
4. **Amalgamate(Name1, Name2):** Transfers all funds from the first customer to the second. Firstly, it computes the sum between the balance of the savings and checking account of Name1. Then, it zeros them out, and adds the previous to the checking balance of Name2.
5. **WriteCheck(Name, Value):** Writes a check against the customer's combined account balance and applies an overdraft penalty if needed. If this total amount of checking and savings is below the value, it takes money out of the checking by value + 1 (as a $1 overdraft penalty).

3 Implementation

Although multiple SmallBank implementations are available for non-partitioned centralized databases, no information is available on how to adapt SmallBank for evaluating geo-distributed systems. As a result, we made a set of design choices to integrate SmallBank into the Detock framework.

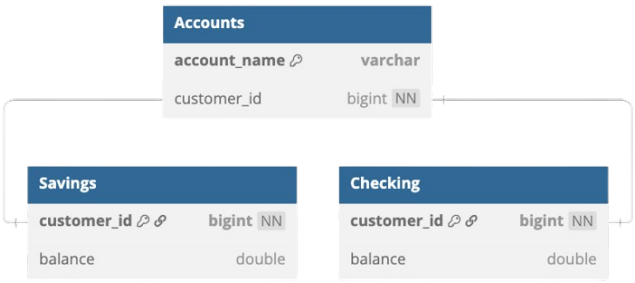

Figure 1: SmallBank's entity relationship diagram (ERD)

3.1 Adapting SmallBank to Detock

The SmallBank workload consists of five distinct transactions whose read or write sets cannot be determined statically as they depend on values from their execution. For example, as shown in Figure 2, the balance transaction needs the *CustomerID* associated with the *CustomerName* before retrieving its account balance. Since Detock does not support dependent transactions [10], we have divided each transaction into multiple phases: one or more helper phases that perform the required reads and a final phase that performs the actual transaction logic. As shown in Table 1, Amalgamate is a three-phase transaction, while the rest are now two-phase transactions. In order to make this possible, we now define each transaction by its type and phase. Firstly, the workload sends a random first phase and binds this information to it. After execution, the system injects the result key into the client (see Listing 1). Then, when sending the next transaction, the client checks for any injected information and sends the second phase if present (see Listing 2). If not, it starts again from the first step. The client repeats this process until the time elapses or an error occurs.

Since both *CustomerIDs* and *CustomerNames* remain constant throughout the workload, the helper phases access only stable rows. Therefore, this does not trigger aborts, a behavior not generally guaranteed in multi-phase transactions, as interleaving with other transactions can lead to conflicts.

Table 1: Phase breakdown of SmallBank transactions in Detock

Txn	Phase Description
Bal	• Retrieve customer ID from name. • Fetch balance using ID.
DC	• Retrieve customer ID from name. • Update checking with deposit.
TS	• Retrieve customer ID from name. • Update savings with amount.
Amg	• Retrieve source customer ID from name. • Retrieve destination customer ID from name. • Fetch balances and transfer amount.
WC	• Retrieve customer ID from name. • Deduct amount and apply penalty if needed.

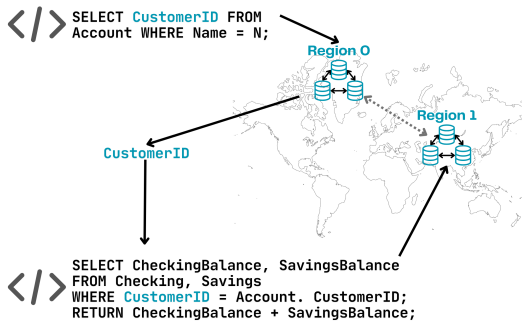

Figure 2: Example of Dependent Transaction Separation

3.2 Partitioning and tables initialization

Multi-partitioned geo-distributed databases rely on deterministic sharding algorithms to distribute records across partitions and geographic location. The primary keys are sharded as follows:

- ***CustomerName:*** We shard using the *murmurhash3* hash algorithm [13], a non-cryptographic hash function suitable for general hash-based lookup which produces deterministic results. This enables the benchmark to determine and store each entry's partition, accurately tracing both single-partition and multi-partition transactions.
- ***CustomerID:*** We shard using the *simple-hashing* algorithm already available in the Detock codebase, also used by TPC-C. Simple-hashing computes the modulo between the integer value of the ID and the number of partitions per region.

This way, during initialization, each partition uses the sharding scheme described above to add only the elements that match the partition ID.

3.3 Transaction distribution

Although the original paper [12] does not address this issue, a random uniform distribution of transactions will not reflect a real-world bank environment, which usually has skewed operation mixes. To address this, as shown in Table 2, we set specific weights for each transaction type. Now, we aim to achieve a more realistic representation and provide a more sensible result.

Table 2: Transaction mix and associated probabilities in the SmallBank benchmark

Transaction	Probability	Max Multi-Home	Max Multi-Partition
Balance	40%	50%	50%
WriteCheck	25%	50%	50%
DepositChecking	15%	50%	50%
TransactionSaving	15%	50%	50%
Amalgamate	5%	33%	33%

3.4 Multi-home and Multi-partition

In geo-distributed databases, data is distributed across multiple geographic regions, referred to as *homes*. Each home is then further divided into multiple *partitions*. Therefore, we define a **multi-home transaction** as a transaction that accesses or modifies data that the system stores in multiple homes. In contrast, a **multi-partition transaction** is a transaction that accesses or modifies data that the system stores in multiple partitions. As a result, we classify transactions into four types based on the data they access:

1.Single-home and Single-partition
2.Single-home and Multi-partition
3.Multi-home and Single-partition
4.Multi-home and Multi-partition

However, since the first phase of the Balance, WriteCheck, DepositChecking, TransactionSaving transactions, as well as the first two phases of the Amalgamate transaction only perform single reads on the accounts table, the system always classifies them as *Single-home and Single-partition*. Unlike those before, the final phase can be configured. Balance, WriteCheck, DepositChecking, and TransactionSaving transactions, checking, and savings accounts can be placed in different regions and/or partitions. Similarly, the system can select the two customers from separate regions and/or partitions for Amalgamate transactions. This allows for configurable multi-home and multi-partition ratios, with upper bounds of 50% and 33%, respectively. As a result, Table 3 shows that evaluations are limited to at most 49.15% multi-home and multi-partition transactions.

Table 3: Configurable bounds for transaction types in the SmallBank benchmark

Type	Minimum	Maximum
Single-Partition	0%	100%
Single-Home	0%	100%
Multi-Home	0%	49.15%
Multi-Partition	0%	49.15%

4 Experimental Setup

In this section, we present the experimental setup and the scenarios in which we perform the benchmarks, as well as the dataset.

4.1 Scenarios

To obtain a comprehensive evaluation, we have designed multiple scenarios that should provide insight into their performance and potential limitations. Each scenario tests a different aspect of the system, such as high contention, network changes, resource use, and workload distribution.

We considered six scenarios, grouped into two categories: Workload Variability and System Conditions.

4.1.1 Workload Variability

In this section, we consider three scenarios. They all assess changes in user behavior patterns and test their potential impact on database performance.

Baseline: This scenario models users performing transactions that require data from multiple regions. To achieve this, we add two configurable parameters: multi-home (MH) and multi-partition (MP). MP is consistently set to 25% (50% of the maximum configurable MP transactions). We run multiple workloads, starting with those containing only single-home transactions (MH = 0%) and gradually increasing to the maximum of multi-home ones (MH = 49.15%). This leads to global coordination, which increases the latency of the transaction.

Skew: This scenario models users performing transactions on a subset of records. To achieve this, we add a configuration parameter named HOT, which simulates a non-uniform access pattern by concentrating the workload transactions on smaller records. We perform experiments ranging from a low contention workload (HOT = 0.0) to a high contention workload(HOT = 1.0), leading to an imbalanced load.

Sunflower: This scenario models users shifting from their usual region to a remote one. We configured clients to target specific regions based on a controllable probability. This experiment uses Region 0 as the target region, with hit probabilities ranging from 10% to 100%. As a result, as the hit probability increases, the client in Region 1 sends more transactions to its remote region (Region 0). In contrast, the client in Region 0 will still target the local region since it is the nearest. To model this behavior, we introduced two configuration parameters:

- SUNFLOWER_TARGET_REGIONS: A list of target regions to be shifted to, with each region becoming active for a duration of $\frac{\text{elapsed time}}{\text{number of target regions}}$.
- SUNFLOWER_HIT_PROBABILITY: A list of percentages specifying the fraction of transactions to send to each target region.

4.1.2 System Conditions

In this section, we consider three scenarios. They all assess changes in the setup's performance and test their potential impact on database performance.

Scalability: This scenario evaluates the system's ability to sustain multiple simultaneous clients. We assess how the number of simultaneous clients impacts total throughput and latency. To achieve this, we run multiple experiments with increasing clients for each region.

Packet loss: This scenario evaluates the resilience of each system to packet loss by measuring transaction performance under varying loss rates. We use a Netem [14], a Linux traffic control tool, to introduce artificial packet loss between the nodes of the setup.

Latency: This scenario evaluates the resilience of each system to network latency by measuring transaction performance under varying latency rates. We use a Netem [14], a Linux traffic control tool, to introduce artificial network latency between the nodes of the setup.

4.2 Dataset

We perform each experiment on the same dataset to ensure consistency and comparability. As shown in Table 4, the dataset consists of three tables, each containing 100K rows and two columns.

Table 4: Schema and size of SmallBank tables in Detock

Table	Rows	Schema
Accounts	100K	Name (24-char), ClientID (0–100K)
Checking	100K	ClientID (0–100K), Balance (0–10K)
Savings	100K	ClientID (0–100K), Balance (0–10K)

4.3 Metrics Considered

Each scenario result consists of a plot containing the database under evaluation compared to four performance metrics, defined as follows:

Throughput: The total number of transactions sent by the workload processesed by the database.

Latency: The 50th (P50) and 90th (P90) percentile latencies across all transactions. P50 represents the median performance, while the P90 captures high-tail performance.

Bytes: The volume of data (in bytes) transmitted between the database nodes.

Cost: The cost of executing the workload's transactions. We calculate the estimated communication cost using a cost model that combines the fixed cost of keeping servers running with the extra cost of sending data between regions [15].

4.4 Setup

The benchmarks are performed using four machines, each featuring 512 GiB (Gigabytes) of RAM and two × AMD EPYC 7H12, each with 64 physical cores.

To simulate a geo-distributed setup with four machines in the same geographical region, we divide them into two logical regions, each having two partitions. In order to replicate the cross-regional delays geo-distributed databases encounter, we introduce an artificial latency of 50 milliseconds between the two logical regions above the already-existent latencies shown in Table 5. Now, we observe a 100ms round-trip latency between regions, closely matching the average P50 latency of 101.29ms reported over the past 365 days between AWS us-west-2 (Oregon) and ap-northeast-1 (Japan) [16].

Furthermore, we deploy the benchmark workload in both logical regions. This reflects real-world usage in which clients access services from both available regions. To simulate locality-aware behavior, each client sends single-home transactions to its nearest region and can send multi-home transactions across regions. This configuration further enables multi-partition transactions, as each region has two partitions.

Table 5: Measured round-trip times between partitions (ms). Inter-region artificial latency ≈ 50ms.

Source \ Target	R1P1	R1P0	R0P1
R0P0	100 + 0.092	100 + 0.377	0.395
R0P1	100 + 0.375	100 + 0.094	–
R1P0	0.096	–	–

5 Experimental Results and Discussion

In this section, we present the results of our experiments and highlight the differences between the geo-distributed

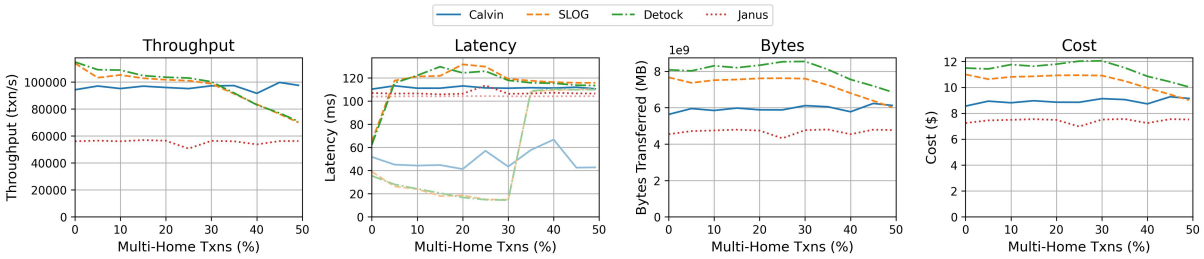

Figure 3: Baseline Scenario Results

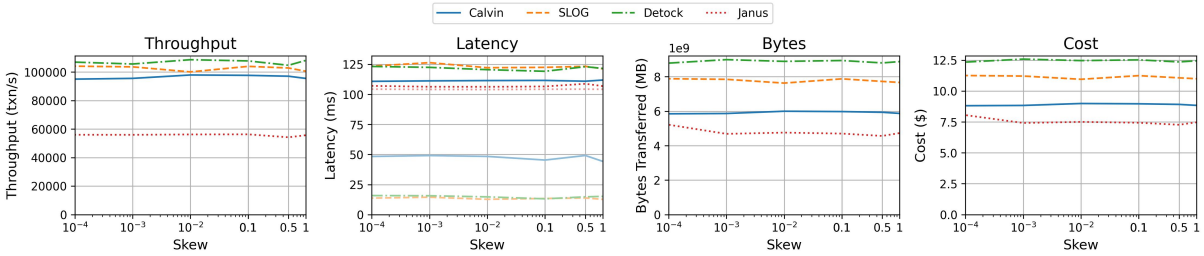

Figure 4: Skew Scenario Results

databases under evaluation.

5.1 Baseline Scenario

Figure 3 shows that Calvin and Janus maintain constant throughput during the experiment, as they do not differentiate between SH and MH transactions. By comparison, Detock and SLOG can benefit from single-home transactions by processing them in parallel in the local regions. Detock, compared to SLOG, achieves a slightly better throughput at almost all MH% levels due to its dependency graphs mechanism and deadlock detection and resolution technique. However, as the MH % sharply increases, an unlikely scenario in a real setting, they incur additional overhead (i.e., dividing the transaction into its region-local components and inserting in the local log of each region), and their advantage disappears, performing even worse than Calvin.

Regarding latency, Janus shows around 100ms, which is equal to the time it takes to complete a full cross-region round-trip. This is due to Janus's biggest weakness: an RTT(Round-Trip Time) for each transaction, regardless of its type. Calvin uses asynchronous replication to achieve a lower P50 latency of about 50ms when transactions remain in a single region. However, the P90 latency increases to around 100ms when cross-region coordination becomes necessary. SLOG performs similarly to Detock, with low latency at low MH% levels, thanks to parallel execution within regions and no need for coordination across regions. As the percentage of multi-home transactions rises above 35%, their latency increases, matching the round-trip time as the system begins coordinating with other regions.

In terms of bytes size and cost, Detock transfers the most bytes per transaction, which is, therefore, more costly, with SLOG being slightly cheaper. In contrast, Calvin and Janus transfer a constant number of bytes, incurring steady costs due to low communication with high latency.

5.2 Skew Scenario

Figure 4 compares the performance of Detock, SLOG, and Janus on a dataset with different percentages of data skew. All databases show constant throughput due to their ordering mechanisms, which ensures high throughput even in high contention settings. Detock, compared to SLOG, shows slightly better throughput due to its deadlock detection and resolution technique. Calvin and Janus perform similarly to the baseline scenario, suffering from the cross-region delays.

In terms of P50 latency, Janus records the highest value at 100ms, corresponding to a full cross-region round-trip, while Calvin follows with a lower latency of approximately 50ms. Detock and SLOG show lower latency due to their ability to process some transactions locally, without requiring communication with the other region.

Detock and SLOG transfer the most bytes per second. On the other hand, Calvin and Janus transfer fewer bytes, achieving lower communication costs. While Calvin and Janus rely on less communication but incur higher latency, Detock and SLOG avoid global coordination for single-home transactions, communicating more within their local partitions.

5.3 Sunflower Scenario

Figure 5 shows that Detock and SLOG perform almost the same in this scenario. At the start, Client 0 and Client 1 send single-home transactions to the region closest to them, simulating users accessing data from their home regions. When the target region hit probability is low, both systems outperform Calvin because they can run transactions in parallel without coordinating across regions, which keeps latency low.

As the hit probability increases, Client 1 sends more transactions to a remote region, while Client 0 sends to its local

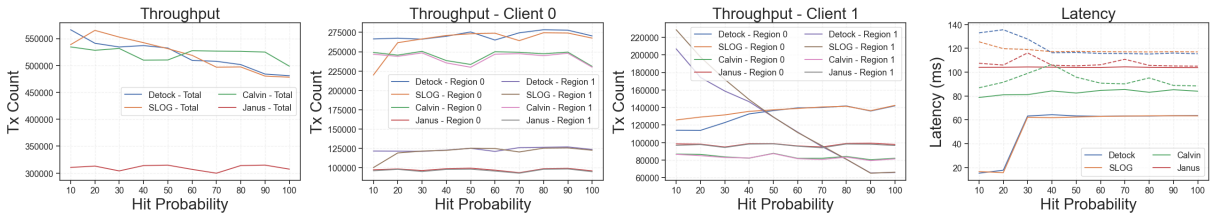

Figure 5: Sunflower Scenario Results

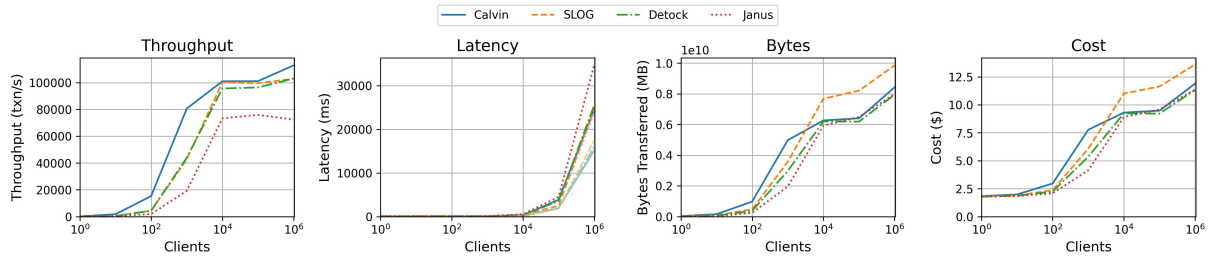

Figure 6: Scalability Scenario Results

region. This setup simulates users from Region 0 traveling to Region 1 and accessing their data from there. Because of the extra communication required, Client 1 handles fewer transactions, which lowers overall throughput. Meanwhile, Client 0 maintains steady performance since it keeps accessing local data. As a result, Region 0 processes more transactions than Region 1, causing an uneven (skewed) load.

However, Janus and Calvin maintain constant performance across all hit probabilities because they treat the system as a single logical region without geographic distinction. As the hit probability reaches 80%, Calvin overtakes Detock and SLOG in throughput. At this point, Client 1 sends all its single-home transactions to what is considered a remote region by SLOG and Detock, introducing cross-region overhead for them. In contrast, Calvin remains unaffected due to its centralized architecture. Janus continues to underperform throughout, reaching only about half the throughput of the other systems because of its high communication overhead.

The P50 latency for Janus and Calvin remains stable across all hit probabilities, as both systems treat the entire deployment as a single logical region, ignoring geographic boundaries. In contrast, Detock and SLOG initially achieve lower P50 latency by executing transactions in parallel within the closest region. However, Client 1 sends more transactions to remote regions as the hit probability increases, causing P50 latency to rise due to the added cross-region communication. Despite this increase, the P90 latency for Detock and SLOG remains stable because the additional overhead from remote single-home transactions is comparable to the latency already introduced by multi-region transactions, which still account for roughly half of the workload and dominate the upper latency bound.

5.4 Scalability Scenario

Figure 6 shows that SLOG and Detock perform similarly, significantly better than Janus. On the other hand, Calvin achieves higher throughput at almost all levels of clients and, therefore, scales better. Detock, SLOG, and Calvin follow an increasing trend up to 10^4 clients, after which it slightly slows until 10^5 and then rises again as the number of clients approaches 10^6. However, for Janus, the throughput increases until 10^3 clients, stays constant until 10^4, and decreases afterward due to its costly cross-region round trip. Detock's and SLOG's throughput slows after achieving 10^4 clients due to complex routing that GraphPlacementTxns in Detock and LockOnlyTxns in SLOG create, serving as a bottleneck until the clients reach 10^6.

The latency, the number of bytes transferred, and the cost all increase as the number of clients and processed transactions increases.

5.5 Packet Loss Scenario

Figure 7 shows that Calvin consistently outperforms Detock, SLOG, and Janus, which show almost identical performance. As packet loss becomes common, throughput declines and latency rises across all the databases we evaluate since coordinating transactions becomes harder, resulting in restarts and aborts. However, Calvin shows better throughput and lower latency than the other systems because it requires less communication to perform transactions. With a 10% packet loss, latency rises to 20,000 ms, making coordination unfesable. As a result, the system's throughput and bytes transferred converge to zero for all the databases. However, as there remains a baseline cost associated with keeping the machine running, the cost becomes constant.

5.6 Latency Scenario

Figure 8 shows that all the evaluated databases show an exponentially decreasing throughput with a corresponding increase in latency. At low delay levels, their throughput is slightly similar to normal conditions. However, as the delay increases, it becomes harder to coordinate transactions

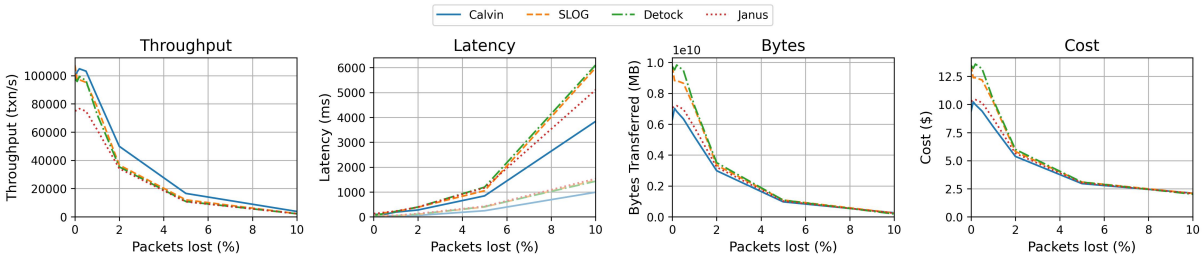

Figure 7: Packet Loss

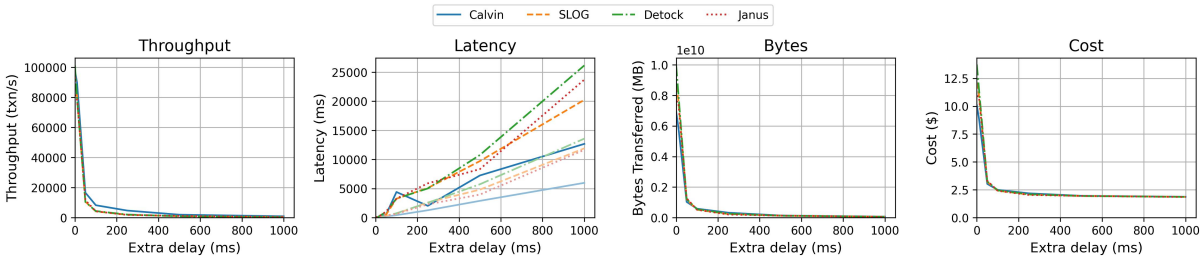

Figure 8: Latency

with some eventually restarting. Detock, SLOG and Janus perform almost identically, both their throughput and latency showing comparable values across all levels while Calvin has slightly better throughput and lower latency due to its lower communication. As the additional delay reaches 1,000 ms, all databases fail to process transactions. Consequently, throughput and total bytes converge to zero. Even if no bytes are sent, the cost becomes constant as the machines keep running, and the baseline cost associated with it remains.

6 Discussion

The previous section presents experimental results that provide a comparison of performance of Detock, SLOG, Calvin, and Janus in different scenarios. However, eventhough the benchmark provides some advantages, it also faces some limitations due to how it handles dependent transactions. Therefore, in this section, we cover both the advantages and limitations of the SmallBank workload.

6.1 Advantages

Customizable Multi-Home and Multi-Partition Parameters: In the basline scenario, our evaluations show that SmallBank's configurable MH and MP transactions provide a great comparison. At first, SLOG and Detock outperform Calvin at low MH levels, but their performance declines as the MH ratio increases, while Calvin maintains a consistently high throughput. However, Janus achieves the lowest throughput with the highest latency, making its required round-trip coordination mechanism its biggest weakness. Although Calvin achieves the highest throughput at high MH% levels, surpassing SLOG and Detock, we rarely encounter such levels in real-world workloads. Under realistic conditions, with low MH% levels, Calvin's throughput is slightly lower than Detock and SLOG. This difference shows the limitations of Calvin's replication strategy and supports the view that Detock and SLOG improve upon its design.

Highlights Communication and Throughput Trade-offs: Excluding Janus, which shows considerably low performance compared to others, Calvin transfers the fewest bytes per transaction across all scenarios, making it the most cost-efficient system. In almost all scenarios, Detock outperforms SLOG due to its replacement of the global ordering layer with a graph-based concurrency control protocol, achieving even higher throughput under high-contention workloads. However, even this slightly better performance comes with an average increase of costs of 10% per workload. Therefore, even if SLOG and Detock get higher throughput in some conditions, the cost overhead can outweigh these gains, making Calvin more convenient for cost-efficient setups.

Controlled Data Hotspot Generation: Even if the throughput is constant for all databases on the skew scenario, Smallbank proves a customizable workload regarding hot records. Using NURand, a non-uniform Random number generator, this workload can send transactions to simulate realistic, skewed access patterns.

6.2 Limitations

Dependent transactions: A major limitation of the SmallBank benchmark workload is its handling of dependent transactions. In particular, the two-phase workaround reduces flexibility, as all but the last phase are fixed, non-configurable both SH and SP transactions. As shown Table in 3, this constraint limits the multi-home MH and MP transaction ratio to a maximum of 49.15%. Even if unrealistic in a real-world setting, this constraint obscures performance differences of geo-partitioned databases under fully multi-home and/or multi-partition workloads.

Transaction Simplicity: Although SmallBank consists of

five distinct transaction types, they lack complex logic. For example, the most common transaction, Balance, performs only a read on two different tables, while DepositChecking, Transaction, and Writecheck provide an extra update. As a result, all the databases show high throughput, allowing them to process transactions efficiently. While this behavior is common in a bank environment, it does not fully represent the more complex workload databases incur.

7 Responsible Research

In this section we reflect on the ethical aspects of the research and discuss the reproducibility of the results.

7.1 Ethical Impact

We use a dataset containing hard-coded Client Names and Client IDs with a randomly generated balance. As a result, we do not use any real customer data or personally identifiable information. Without data privacy, consent, or bias issues, this research does not raise any ethical concerns.

7.2 Reproducibility

Our research adheres to the FAIR (Findability, Accessibility, Interoperability, and Reusability) guidelines for scientific data management and stewardship [17].

We implemented the SmallBank workload and conducted all benchmark experiments on top of Detock's publicly available GitHub code [18]. Additionally, we published the modifications and configuration files of each experiment in the Delftada repository [19], making it ***findable*** and ***accessible*** to future fellow researchers who want to pursue more research on this topic.

Our experimental results are both ***interoperable*** and ***reusable*** since our workload employs deterministic functions to generate transactions with fixed seeds for randomness and consistent configuration files. We also include documentation and setup instructions to install the required dependencies, configure the environment, and execute the experiments as well as a series of scripts to gather the results from the setup nodes, generating visualizations on the reported measurements.

8 Future work

Additionally, there are paths future researchers can pursue to improve the work presented in the previous sections. Therefore, we develop a series of ideas that build upon this research.

Limited Setup Configuration: The setup on which we applied the workload is limited. However, we could increase both the number of partitions and the number of regions to replicate a wider geographic distribution. Having different inter-region latencies and a more complex setup, we could analyze behavior under more realistic cloud deployment conditions.

Two-phase transactions: We can implement several improvements to the actual work. For example, we can implement Deneva's version of Calvin (using the OLLP protocol to obtain an initial estimate of the access set via a reconnaissance query) to eliminate the need of multi-phase transactions. Additionally, we could create a comparison of performance between the two-transaction workaround and the transactions OLLP protocol to asses how this impacts performance.

Complex Transaction Logic: Since the Smallbank workload is relatively simple, all databases showed high throughput across the scenarios. As a result, we could increase the complexity of the existing workload. By introducing a separate table for accounts for children, we could implement more transaction types, such as creating accounts, transferring funds to them, and later deleting them. This extra functionality can evaluate performance in a situation that SmallBank lacks, such as handling inserts and deletions.

9 Conclusions

In this paper, we investigated the behavior of Detock, SLOG, Calvin, and Janus, some of the newest modern databases aimed at minimizing inter-region communication. We studied their performance across six scenarios designed to stress different system parts. For each configuration, we discussed the strengths and limitations of the databases. Overall, the SmallBank benchmark shows that Detock and SLOG perform best, having high throughput and low latency. In contrast, Calvin proves more effective in environments with a high proportion of multi-home transactions and shows a lower operation cost than Detock and SLOG throughout the experiments. In contrast, Janus shows the weakest performance across all scenarios.

Consequently, we believe that SmallBank offers several advantages: It can generate a data hotspot and is customizable for multi-home and multi-partition transactions, which makes it a valuable comparison for geo-distributed databases. However, further updates on this workload, such as eliminating the two-phase transactions, scaling the setup on which we perform measurements, or adding more complex transactions, would better assess the system's performance.

References

[1] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, page 205–220, New York, NY, USA, 2007. Association for Computing Machinery.

[2] Yugabyte Inc. Top 5 us bank modernizes bill payment with yugabytedb. https://resources.yugabyte.com/hubfs/yugabytedb-top-5-us-bank-case-study.pdf, 2022. Accessed: 2025-06-06.

[3] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. Tao: Facebook's distributed

data store for the social graph. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, page 49–60, USA, 2013. USENIX Association.

[4] Scott T. Leutenegger and Daniel Dias. A modeling study of the tpc-c benchmark. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, page 22–31, New York, NY, USA, 1993. Association for Computing Machinery.

[5] 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.

[6] 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.

[7] Brecht Vandevoort, Bas Ketsman, Christoph Koch, and Frank Neven. Robustness against read committed for transaction templates. *Proc. VLDB Endow.*, 14(11):2141–2153, July 2021.

[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 '12, page 1–12, New York, NY, USA, 2012. Association for Computing Machinery.

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

[10] Cuong D. T. Nguyen, Johann K. Miller, and Daniel J. Abadi. Detock: High performance multi-region transactions at scale. *Proc. ACM Manag. Data*, 1(2), June 2023.

[11] 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.

[12] Mohammad Alomari, Michael Cahill, Alan Fekete, and Uwe Rohm. The cost of serializability on platforms that use snapshot isolation. In *2008 IEEE 24th International Conference on Data Engineering*, pages 576–585, 2008.

[13] Austin Appleby. Murmurhash3. https://github.com/aappleby/smhasher/blob/master/src/MurmurHash3.cpp, 2011. Accessed: 2025-06-10.

[14] Stephen Hemminger and contributors. *tc-netem(8) — Network Emulator*. Linux Foundation, February 2025. Linux manual page for network emulation with netem.

[15] Dan Graur, Oto Mraz, Muyu Li, Sepehr Pourghannad, Chandramohan A. Thekkath, and Ana Klimovic. Pecan: Cost-efficient ml data preprocessing with automatic transformation ordering and hybrid placement. In *Proceedings of the 2024 USENIX Annual Technical Conference (USENIX ATC 24)*. USENIX Association, 2024.

[16] Aws region latency matrix. https://www.cloudping.co/. Accessed: 2025-06-10.

[17] M. Wilkinson, M. Dumontier, and I. et al. Aalbersberg. The fair guiding principles for scientific data management and stewardship. *Sci Data*, 3:160018, 2016.

[18] UMD DSLAM Group. Detock: High Performance Multi-region Transactions at Scale. https://github.com/umd-dslam/Detock, 2023. Accessed: 2025-06-14.

[19] Cirtog Filip. SmallBank Benchmark Implementation in Detock. https://github.com/delftdata/Detock/tree/smallbank, 2025. Accessed: 2025-06-14.

A Appendix

Listing 1: Sending a transaction based on the previous one.

```
Transaction* txn = new Transaction();
if (b_txn != nullptr) {
 CHECK(b_txn->keys_size() == 1);
 memcpy(&c_id, b_txn.data(),sizeof(int));
 Balance(*txn, pro, 2);
}

if (d_txn != nullptr) {
 CHECK(d_txn->keys_size() == 1);
 memcpy(&c_id, d_txn.data(),sizeof(int));
 DepositChecking(*txn, pro, 2);
}

if (t_txn != nullptr) {
 CHECK(t_txn->keys_size() == 1);
 memcpy(&c_id, t_txn.data(),sizeof(int));
 TransactionSaving(*txn, pro, 2);
}

if (w_txn != nullptr) {
 CHECK(w_txn->keys_size() == 1);
 memcpy(&c_id, w_txn.data(),sizeof(int));
 Writecheck(*txn, pro, 2);
}

if (a1_txn!=nullptr && a2_txn==nullptr) {
 a2_txn = a1_txn
 a1_txn = nullptr;
 Amalgamate(*txn, pro, 2);
}

if (a1_txn!=nullptr && a2_txn!=nullptr) {
 CHECK(a1_txn->keys_size() == 1);
 CHECK(a2_txn->keys_size() == 1);
 memcpy(&id1, a1_txn.data(),sizeof(int));
 memcpy(&id2, a2_txn.data(),sizeof(int));
 Amalgamate(*txn, pro, 3);
}
```

Listing 2: Injecting the previous workload for the next transaction.

```
if (workload_->name() == "smallbank") {
 if (dependency == FIRST_PHASE) {
 // Record results returned by 1st phase.
  SBWorkload* workload = workload_.get();
  if (trn_type == BALANCE)
    workload->b_txn = txn;
  if (trn_type == DEPOSIT_CHECKING)
    workload->d_txn = txn;
  if (trn_type == TRANSACTION_SAVING)
    workload->t_txn = txn;
  if (trn_type == AMALGAMATE)
    workload->a_txn = txn;
  if (trn_type == WRITECHECK)
    workload->w_txn = txn;
 }
}
```

Listing 3: Computing the home

```
Metadata SBI::Compute(const Key& k) {
 int h;
 if (k.size() == 26) {
    string name =  k.data();
    h = (murmurhash3(name) / np) % nr;
 } else {
    int id = k.data();
    h = (id / np) % nr;
 }
 return Metadata(h);
}
```

Listing 4: Computing the partition

```
int Sharder::Compute(const Key& k) const{
  if (key.size() == 26) {
    string name = k.data();
    return murmurhash3(name) % np;
  } else {
    int id = k.data();
    return id % np;
  }
}
```

Listing 5: Loading tables for each partition

```
mt19937 rg;
uniform_int_distribution<> bal(1, 10000);

for (int id = start; id < end; id++) {
    string n = "Client" + to_string(id);
    n.resize(24, ' ');  // Pad if shorter
    int h = murmurhash3(n);
    if (h % np == partition_) {
        accounts.Insert({
          MakeFixedTextScalar<24>(n),
          MakeInt32Scalar(id)
        });
    }
    if (id % np == partition_) {
        checkings.Insert({
          MakeInt32Scalar(id),
          MakeInt32Scalar(bal(rg))
        });
        savings.Insert({
          MakeInt32Scalar(id),
          MakeInt32Scalar(bal(rg))
        });
    }
}
```