Benchmarking Geo-distributed Databases

Evaluating Performance using the Product-Parts-Supplier Workload

Eduard-Alex Mihai[1]

Supervisors: 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: Eduard-Alex Mihai
Final project course: CSE3000 Research Project
Thesis committee: Asterios Katsifodimos, Oto Mráz, Koen Langendoen

Abstract

Existing evaluations of geo-distributed databases still rely almost exclusively on standard limited workloads such as TPC-C and YCSB+T, which reveal little information about the true cost of wide-area coordination. In this paper, we present a configurable benchmarking framework built around the Product-Parts-Supplier (PPS) workload, and use it to evaluate four representative systems that support geo-distributed transactions: Calvin, SLOG, Detock, and Janus. The experiments run across six realistic and insightful scenarios that vary the transactional load, contention, client count, regional bias, network latency, and packet loss.

The results uncover clear design trade-offs between the systems and demonstrate that our new framework is capable of filling several evaluation holes left by the standard workloads. Our framework introduces important features such as dependent transactions that may abort and retry, longer and tunable read/write sets, and fine-grained control over which regions will participate in the commit. However, the PPS workload comes with some limitations, and thus, the framework does not cover every case. As a consequence, for now, it should complement TPC-C and YCSB+T, not replace them.

1 Introduction

In the modern era of cloud computing and real-time global services, geo-distributed databases have become essential to many critical large-scale applications [1, 2]. These systems replicate and partition data across multiple geographical regions to provide high durability and significantly improve the latency by storing the data closer to the users. However, this distributed infrastructure comes with several tradeoffs. Given the high communication delays between different servers, it becomes increasingly challenging to synchronize transactions that access data scattered across multiple regions [3].

Researchers have proposed several database systems, including Calvin [4], SLOG [5], Detock [6], and Janus [7], to improve the transaction throughput and reduce the latency in geo-distributed environments. For instance, Detock uses a novel graph-based deterministic deadlock resolution protocol that allows the system to efficiently process multi-region transactions. Despite these innovations, public performance evaluations of these systems are still limited, especially under complex workloads and diverse deployment conditions. Most studies, including those presented in the original system papers, rely only on standard workloads such as TPC-C [8] and YCSB+T [9], which are known to have a limited scope when it comes to distributed transactional databases [10]. TPC-C is highly partitionable, with most transactions accessing only a single partition, which limits the ability to stress the inter-region communication and coordination overhead. YCSB+T, while extending the original YCSB workload to include multi-key transactions, keeps a simplistic key-value data model and lacks important features such as foreign keys, join operations, and relational constraints.

To address these limitations, researchers now complement TPC-C and YCSB+T with more complex workloads such as Product-Parts-Supplier (PPS). Harding et al. [11] demonstrated that PPS can uncover performance bottlenecks and communication inefficiencies that simpler benchmarks miss. However, most studies typically use PPS under a narrow set of configurations, making it difficult to fully understand the system behavior under different conditions.

This paper aims to address the lack of comprehensive evaluations by investigating *how modern database systems perform under the PPS workload in geo-distributed settings, and how workload characteristics and environmental factors influence throughput, latency, abort rates, network utilization, and operational cost*.

The main contribution of this work is a configurable benchmarking framework based on the PPS workload and designed to evaluate geo-distributed databases under a variety of realistic scenarios. In addition, we provide a comparative evaluation of several representative systems. The findings aim to inform about the practical performance limits and the potential bottlenecks of these systems and motivate the need for more comprehensive benchmarks.

The remainder of this paper proceeds as follows. Section 2 provides the necessary background for this work, including the important characteristics of the evaluated databases and a formal description of the PPS workload. Then, Section 3 describes the implementation of the benchmarking framework, and Section 4 presents the experimental setup and analyzes the results across various scenarios. In Section 5 we discuss the steps taken to ensure the reproducibility of our experiments, and in Section 6 we reflect on the advantages and limitations of using PPS as a benchmark. Finally, Section 7 concludes with the key findings, summarizes the contributions, and outlines potential directions for future research.

2 Background

In this section, we introduce the database systems used in our evaluation (Section 2.1), and we present the key features of the PPS workload that represent the foundation of our benchmarking framework (Section 2.2).

2.1 Evaluated Systems

This study evaluates four database systems that support globally distributed transactions: Calvin, SLOG, Detock, and Janus, using the benchmarking framework we implemented. Each system has its unique deployment strategy and concurrency control approach for handling distributed transactions.

We selected these four systems because they all support geo-distributed transactions and share comparable architectural principles, such as deterministic execution and graph-based serialization, which aim to reduce the runtime wide-area communication. Despite these foundational similarities, they have clear differences that make each system more suitable for particular workload characteristics or deployment scenarios. Another important aspect is that the Detock codebase[1] provides a unified implementation of all four systems. So, Calvin, SLOG, and Janus were re-implemented within

[1] https://github.com/umd-dslam/Detock

the Detock framework to share a common storage engine, communication layer, and local consensus protocol [6]. This shared infrastructure ensures that the performance differences from our evaluation come from each system's core design choices rather than unrelated implementation details.

Deterministic Scheduling. Calvin, SLOG, and Detock are based on *deterministic scheduling*, an alternative to traditional protocols that use atomic commitment [12]. These deterministic database systems avoid the need for runtime coordination between different servers by predefining the execution order of the transactions. This design reduces the communication overhead during transaction execution, which offers notable performance benefits, especially in the presence of inter-region delays. However, this type of locking protocol requires complete knowledge of each transaction's read and write sets in advance. As a result, these systems face challenges when handling *dependent transactions*, namely transactions for which the accessed records are not known beforehand, but must be discovered as the execution progresses.

Calvin [4] demonstrated the practical feasibility of deterministic transaction processing with near-linear scalability. The system is organized into three separate layers. First, the *sequencing layer* receives the transactional requests from the clients, batches them using short epochs, and ensures consistent ordering among the replicas. Then, the *scheduling layer* coordinates the transaction execution by acquiring the locks deterministically according to the defined global order and by passing the transactions to the pool of workers. Finally, the *storage layer* handles the physical data access.

SLOG [5] is built on top of Calvin by optimizing for data locality when used in geo-distributed settings. It introduced the notion of *single-home (SH) transactions* that access data in a single region, and thus can be processed with very low latency without being passed to the sequencing layer. However, the *multi-home (MH) transactions* still follow Calvin's sequencing model and must adhere to a global order.

Detock [6] is also based on Calvin's architecture, but it removes the need for global ordering by using a deadlock resolution protocol via a *dependency graph*. This approach allows both single-home and multi-home transactions to be scheduled deterministically at each region. In this way, all participating regions construct independently the same dependency graph, and the transactions can be run locally without the need for cross-region communication once all transaction components have been received.

Unified Consensus and Concurrency Control. *Janus* [7] integrates the concurrency control mechanism (used for transaction consistency) with the consensus protocol (used for replication) into a single layer. The key idea is that both components work by ensuring that the execution history of the transactions is equivalent to a sequential order, which can be verified by defining a *serialization graph*. Janus can commit and replicate the transactions within a wide-area network (WAN) round-trip under low contention, and requires at most one additional round-trip in case of conflicts.

2.2 Product-Parts-Supplier Workload

The Product-Parts-Supplier (PPS) workload is a well-known relational structure that simulates a realistic supply chain management system. It captures the interactions between the existing products, the parts that make up those products, and the suppliers that provide those parts.

PPS has been used as a workload in several popular database benchmarking studies, although typically in a limited capacity. For example, Harding et al. [11] utilized the PPS workload primarily to evaluate the system scalability by varying only the number of client machines, while keeping the other settings fixed. Serafini et al. [13] applied the PPS in their evaluation, but they included only read-only transactions since their goal was to assess the effectiveness of their dynamic partitioning scheme rather than to evaluate the full concurrency behavior of the system.

Our benchmarking framework is built upon a set of tables and transaction types drawn from these previous uses of the PPS workload. Specifically, the schema includes three core entities, *Products*, *Parts*, and *Suppliers*, along with two tables that capture their many-to-many relationships. A product can contain multiple parts, and a part can belong to multiple products. In the same way, a supplier can provide multiple parts, and a part can be provided by multiple suppliers. Figure 1 shows the relational schema of the PPS workload.

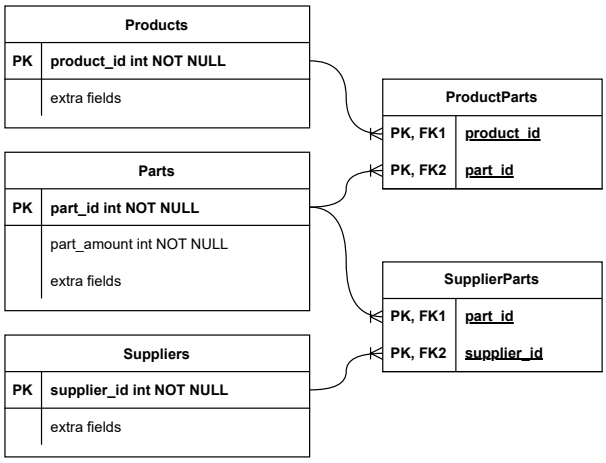

Figure 1: Product-Parts-Supplier Entity Relationship Diagram.

The PPS workload has transactions with complex data dependencies that can lead to contention challenges when multiple transactions run concurrently. This complexity makes the workload suitable for evaluating how database systems handle the coordination and conflict resolution. The workload includes several types of transactions, namely:

- *OrderProduct*. Given a `product_id`, this transaction will first collect the parts associated with the given product and then decrement the inventory amount for those parts. Importantly, this is a dependent transaction, since we do not know which parts will be updated in advance.
- *GetPartsByProduct*. Given a `product_id`, this transac-

tion will retrieve the list of parts that are currently associated with the given product.

- *UpdateProductPart*. Given `product_id`, `part_from`, and `part_to`, this transaction will update a part associated with the given product by replacing the part identified with `part_from` with the latter part `part_to`.
- *GetPart*. Given a `part_id`, this transaction will retrieve the current inventory amount of the given part, along with any existing extra information.
- *GetProduct*. Given a `product_id`, this transaction will retrieve any extra information about the given product.

3 Benchmark Implementation

This section presents the implementation of our benchmarking framework, designed to evaluate the performance of geo-distributed transactional systems based on the PPS workloads. We discuss here key design choices, such as the integration of the benchmarking framework with the evaluated databases (Section 3.1), the support for dependent transactions (Section 3.2), the custom partitioning and home assignment schemes that offer precise control over transaction configurations (Section 3.3), and the transactional mix of the workload (Section 3.4).

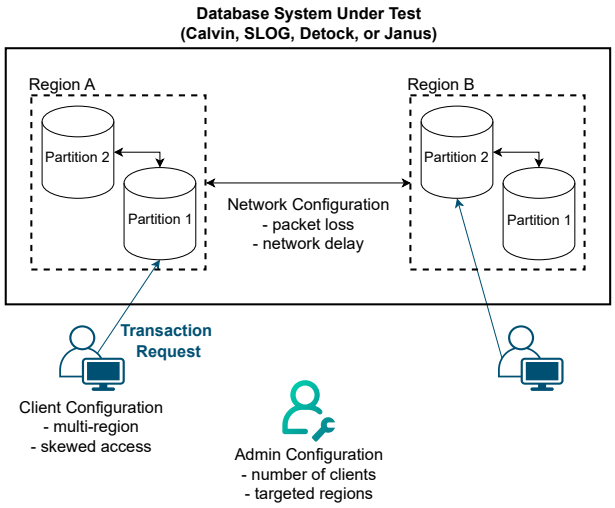

Figure 2: Architecture of the Benchmarking Framework, together with all the Tunable Parameters.

3.1 Integration with Different Databases

The databases we evaluate follow a common setup, where data is partitioned across different servers, and each partition is replicated in different geographically distributed regions to ensure high availability and low latency by placing the data closer to the users. Our benchmark spawns client processes in different regions and can target specific partitions within any region. The architecture of our benchmarking framework and the way it interacts with the databases is shown in Figure 2.

As stated in Section 2.1, we implemented the benchmark within the Detock codebase to offer a unified testing environment for all the evaluated systems. This integration also allows us to define key components such as table schemas, table loaders, and stored procedures in a single place and use them consistently across all databases.

3.2 Dependent Transactions

As mentioned in Section 2.1, all the evaluated databases need to know the complete set of accessed records for each transaction to either deterministically order them or construct the serialization graph. Among the transactions of the PPS workload, only *OrderProduct* is a dependent transaction since it must first determine which parts are associated with the product before any updates can take place.

In its original design, Calvin was implemented to support dependent transactions by using a scheme called Optimistic Lock Location Prediction (OLLP) that makes use of a reconnaissance query to perform all the necessary reads to compute the complete read/write set before the actual execution begins. Unfortunately, this mechanism is not currently implemented in the Detock framework. As a solution, we will use a similar approach by splitting the *OrderProduct* transaction into *two client-side phases*. In the first phase, the client fetches the current list of parts associated with the given product. Then, in the second phase, the client issues the update request with the complete set of accessed records. It is important to note that when the second phase detects any changes in the reads done by the first phase, it will self-abort.

3.3 Partitioning and Home Assignment Schemes.

Our benchmark supports custom partitioning schemes. In our setup, we partitioned the data across nodes in a round-robin fashion based on the record identifier (`product_id`, `part_id`, `supplier_id`) for the core entities, and the `product_id` for the two additional relationship tables. We say that a transaction is *single-partition* if it accesses data from only one partition, and *multi-partition* if it spans multiple partitions.

In addition to partitioning, SLOG and Detock use a couple of optimizations based on the concept of *home regions*. In short, each record is assigned to exactly one geographical region, which is called the home region of this record and is responsible for coordinating the access to that record. Both systems assume that, theoretically, within each region, data is partitioned locally regardless of which region owns the data. This means that a single partition can hold records whose home regions are different. Similar to partitioning, we can classify the transactions into *single-home* and *multi-home*. Our implementation supports custom home assignment schemes. In our evaluation, we assign home regions in a round-robin fashion within each partition.

In our workload, only the second phase of the *OrderProduct* transaction can be configured to be either single-partition or multi-partition, and independently, single-home or multi-home, depending on which parts are involved. The other transactions are simpler, since they either access only one record or all the accessed records fall within a single partition and home region, according to the chosen partitioning

and home assignment scheme. The main challenge in configuring the second phase of the *OrderProduct* transaction is that we do not know which parts will be accessed in advance. To overcome this, when creating the tables, we split the products into four groups based on the distribution of their parts:

- *Category I*. All parts are located in the same partition and have the same home region as the product itself.
- *Category II*. All parts are located in the same partition as the product, but belong to different home regions.
- *Category III*. All parts have the same home region as the product, but are located in different partitions.
- *Category IV*. The parts are located in different partitions and also belong to different home regions.

By organizing the parts into these four categories, the benchmarking framework can dynamically generate *OrderProduct* transactions with specific characteristics. For example, if we want to generate a single-home multi-partition transaction, we select a product from *Category III*.

3.4 Transactional Mix

Section 2.2 introduced the type of transactions that we consider in our benchmarking framework. While all these transactions reflect realistic business operations, such as updating a product or retrieving a part, not all of these types are well-suited for evaluating geo-distributed behavior. In particular, as stated in the previous subsection, only the *OrderProduct* transaction has an access pattern that can span multiple partitions and regions, making it the only meaningful operation that can test the servers' coordination.

As a consequence, we used a transactional mix that favors the *OrderProduct* type. In addition, we assigned greater weights to the *UpdateProductPart* type, since it can interfere with two-phase dependent transactions, and to the *GetPartsByProduct* type, since it accesses multiple records, making it more relevant for evaluating concurrency behavior. In contrast, the simple read-only transactions *GetPart* and *GetProduct* have lower priority. Table 1 summarizes the mix.

Transaction Type	Ratio
OrderProduct	80%
GetPartsByProduct	8%
UpdateProductPart	8%
GetPart	2%
GetProduct	2%

Table 1: PPS Transactional Mix used in our Experiments.

4 Experimental Setup and Results

This section outlines our experimental methodology and key findings. Section 4.1 describes our deployment setup and the methodology used for performance measurement, and Section 4.2 presents the results obtained under a range of different experimental scenarios.

4.1 Deployment and Metrics Collection

We conducted all experiments on a dedicated 4-node high-performance cluster located in our university. Each node is equipped with dual AMD EPYC 7H12 processors, 256 hardware threads, and 503 GiB of RAM. The nodes are interconnected via 10 Gigabit Ethernet.

In our experiments, we simulated a two-region deployment by using all four physical nodes and injecting artificial network delay between them. Both clients and servers are deployed as Docker containers to ensure a clean and reproducible environment. Figure 2 illustrates the setup. The data is divided into two partitions, and each logical region consists of two server containers that hold a complete copy of both partitions. The cluster delivers a natural intra-region round-trip time (RTT) of roughly 0.15 ms, and we set the inter-region RTT to 100 ms (Table 2).

	A-P1	A-P2	B-P1	B-P2
A-P1	—	0.15 ms	100 ms	100 ms
A-P2	—	—	100 ms	100 ms
B-P1	—	—	—	0.15 ms
B-P2	—	—	—	—

Table 2: Round-trip times between all unordered pairs of machines. We uniquely identify the machines with the label *Region-Partition* (e.g., *A-P1* is the partition 1 within the region A).

Unless stated otherwise, the clients are uniformly distributed across the two regions. To avoid overwhelming or underdriving the systems, we first explore the effects of the number of clients using the scalability test from Section 4.2.4. We identify for each system the point where additional clients would no longer increase the throughput, and use that client count in all remaining scenarios.

For performance evaluation, each client container collects local metrics during execution, which are then aggregated by a centralized admin. The metrics we focus on are throughput, latency, abort rates, and bytes transferred, and we also estimate the operational cost. Throughput, latency, and abort rates are measured directly by the application logic, while the number of transferred bytes is collected using system-level network monitoring tools. Additionally, we provide a simplistic estimation of the operational cost by monitoring the resource utilization within the server containers and tracking the overall network usage. The formula we use for the hourly cost is $C = N_{\text{machine}} * P_{\text{machine}} + V_{\text{traffic}} * P_{\text{traffic}}$, where N_{machine} is the number of server containers, P_{machine} is the on-demand hourly price for the each used machine, V_{traffic} is the cross-region traffic volume, and P_{traffic} is the data transfer price.

4.2 Evaluation Scenarios and Results

We now describe the experimental scenarios used in our evaluation. We designed these scenarios to systematically observe the impact of different system configurations on the transactional performance of the database systems. Each scenario is strongly connected with the tunable parameters shown in Figure 2, which control the network conditions, the clients' placement, and the transactional load.

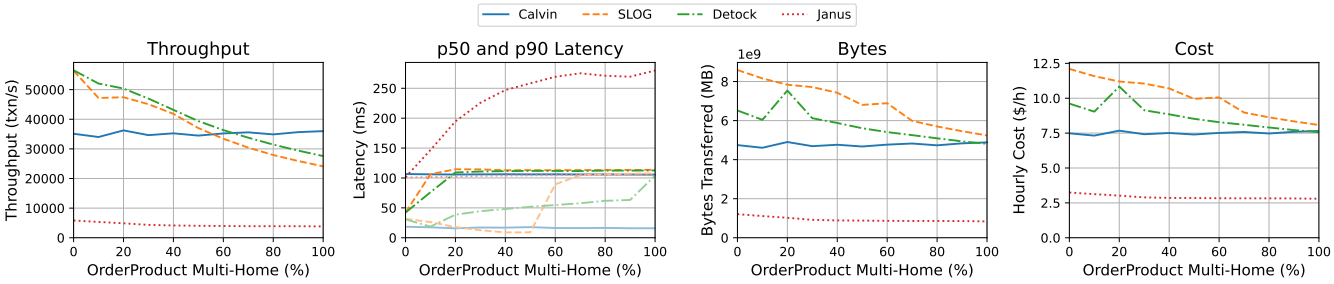

Figure 3: Results of the Baseline Scenario.

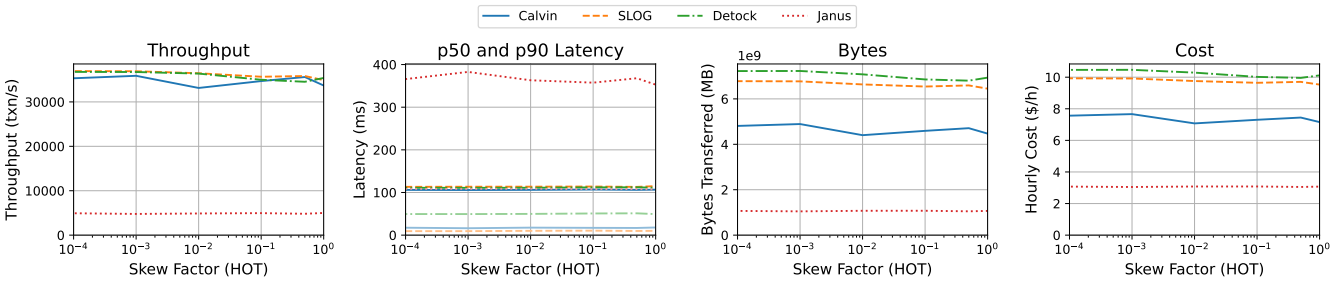

Figure 4: Results of the Skew Access Scenario.

4.2.1 Baseline Scenario

In the baseline scenario, we aim to understand how each concurrency control protocol performs under standard conditions in the two-region deployment with no artificial skew, packet loss, or extra delay besides the default link latencies shown in Table 2. Specifically, we vary only the fraction of *multi-home transactions* among the *OrderProduct* type while keeping the proportion of multi-partition transactions constant at 50%. This scenario allows us to isolate the cost of coordination across different geographical regions. Figure 3 shows the metrics collected for each protocol: the throughput, the median latency ($p50$), along with the 90^{th}-percentile latency ($p90$), the bytes transferred, and the hourly operational cost.

Among all systems, Janus has the lowest throughput and the highest latencies. This is primarily because Janus does not have the notion of multi-home transactions, and routes every commit through its integrated consensus path, which requires cross-region coordination even for transactions that could be executed locally. As a consequence, the throughput stays constant, and even the single-region transactions need at least one WAN round-trip, which is visible in the plot as the $p50$ latency sits near the inter-region RTT ($\approx$100 ms). In case of conflicts, Janus needs an additional wide-area round-trip, so $p90$ latency exceeds 200 ms, and even slightly increases with larger MH fractions, since this inherently increases the contention, as the regions will contend for the same products.

Calvin also does not have the notion of multi-home transactions, and thus, the throughput stays almost constant. It performs worse than SLOG and Detock when the proportion of multi-home transactions is low, but it overtakes them as the proportion exceeds roughly 60%. This is because Calvin relies on a deterministic global sequencer that orders all transactions. When most transactions are local, Calvin's overhead for global ordering becomes unnecessary and costly, but as the proportion of multi-home transactions increases, Calvin's global sequencer can coordinate them efficiently and get a better performance than SLOG and Detock, which incur a significant cost of handling multi-home transactions.

4.2.2 Skewed Access Scenario

In the skewed access scenario, we study how each protocol handles the contention caused by an uneven access pattern. We fix the workload composition so that half of the *OrderProduct* transactions are multi-home, and similarly, half of them are multi-partition. To introduce skew, we use the NURand distribution to define so-called *hot records* that will be accessed more often depending on the *skew factor* [14].

As shown in Figure 4, the curves remain nearly flat for all four systems across the entire range of skew factors. For the deterministic designs (Calvin, SLOG, Detock), this is expected since each replica already knows the global order before the execution begins, so a hot key cannot block related transactions. The skew also has little effect on Janus, showing that the system's bottleneck is given by the WAN round-trip.

Figure 6 reveals the hidden cost of the skew. The protocols themselves don't abort transactions, since Calvin, SLOG, and Detock commit every transaction in the predetermined order, and Janus commits according to the serialization graph. Therefore, all aborts we observe come from the way we execute the dependent transaction *OrderProduct*. We split it into a read-only phase that retrieves the parts currently associated with the given product, and a write phase that performs the updates. If any of those parts change between those two phases, the server aborts the second phase and signals the client to retry the whole transaction starting from the first

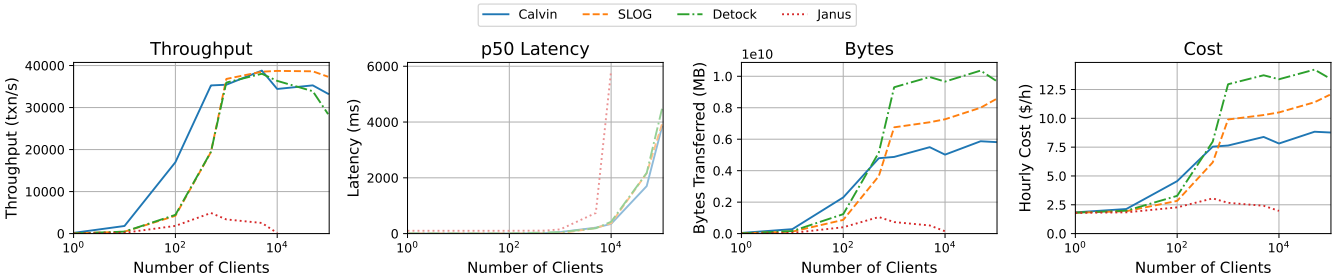

Figure 5: Results of the Scalability Scenario.

phase. The presence of hot records raises the chance that the second phase's validation fails, which explains why abort rates increase with the skew for all systems. However, the increase is small, peaking at only 2.5% among all systems, and leaving throughput and latency practically unchanged.

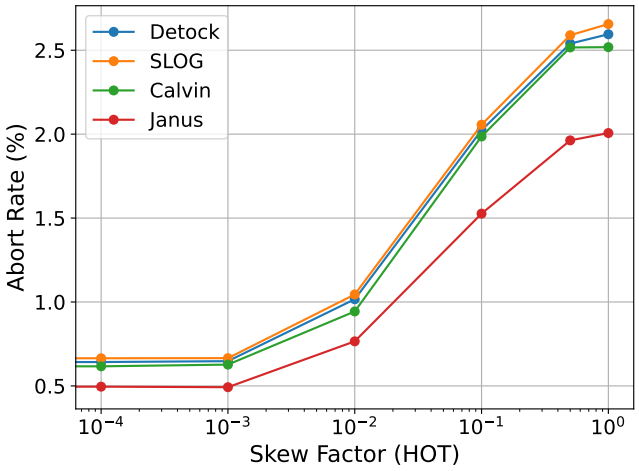

Figure 6: Abort Rates in the Skew Access Scenario.

4.2.3 Sunflower Topology Scenario

The sunflower experiment analyzes how the protocols react when one region turns into a central hub. This is a common occurrence in the real world, where deployments rarely receive perfectly balanced traffic. A viral event, a seasonal sale, or an outage in a neighboring zone can redirect requests into one data centre while the others stay idle. To reproduce this, we perform a 100-second run, where we start with balanced traffic, and then linearly increase the share of transactions whose home region is *Region A* (or, for multi-home transactions, include *Region A* in the home set) with 10% every 10 seconds, until it receives the entire load. Throughout the experiment, we keep the workload mix fixed at 50% multi-partition and 50% multi-home transactions, and leave the default RTTs unchanged, so any throughput change reflects only the growing regional imbalance.

Figure 7 plots the resulting throughput. At the start, when traffic is evenly split, all systems have their baseline throughput. As the bias grows, SLOG and Detock are the only ones that experience a decrease in throughput. Once more requests land in the same region, the single-home fast path overwhelms the replica while its counterpart in *Region B* sits mostly idle, so the aggregate throughput drops.

In contrast, Calvin's throughput stays approximately the same. In the Detock codebase, Calvin's global sequencer uses a replication based on a primary replica by asynchronously sending every transactional input (both the single-home and multi-home transactions) to a dedicated region that broadcasts the batches in order. Since the primary replica already acts as a central point, increasing the bias doesn't have an effect on the overall throughput. Similar to the skew access scenario (4.2.2), Janus once again shows the lowest absolute throughput, but it is practically insensitive to the bias. Every commit must gather a WAN approval, so moving clients from *Region A* to *Region B* doesn't affect the throughput.

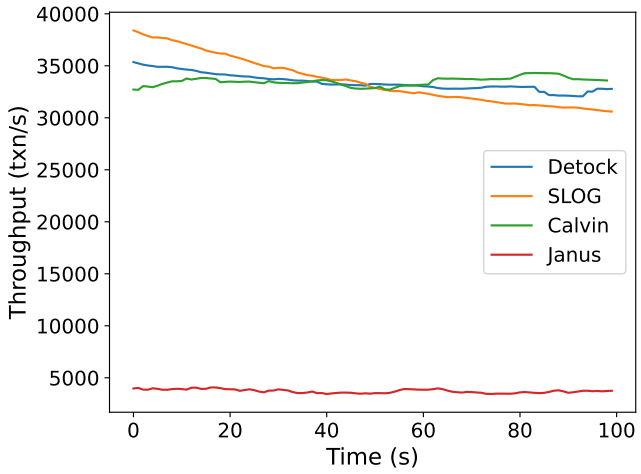

Figure 7: Throughput in the Sunflower Topology Scenario.

4.2.4 Scalability Scenario

The scalability scenario focuses on evaluating how each protocol performs as we increase the number of clients, while keeping the workload composition fixed at its standard configuration with 50% multi-home transactions, 50% multi-partition transactions, and no added skew. Figure 5 shows how SLOG, Detock, and Calvin all have similar improvements in throughput as the load increases, which demonstrates that these systems have good scalability given a balanced transactional workload. Janus, on the other hand,

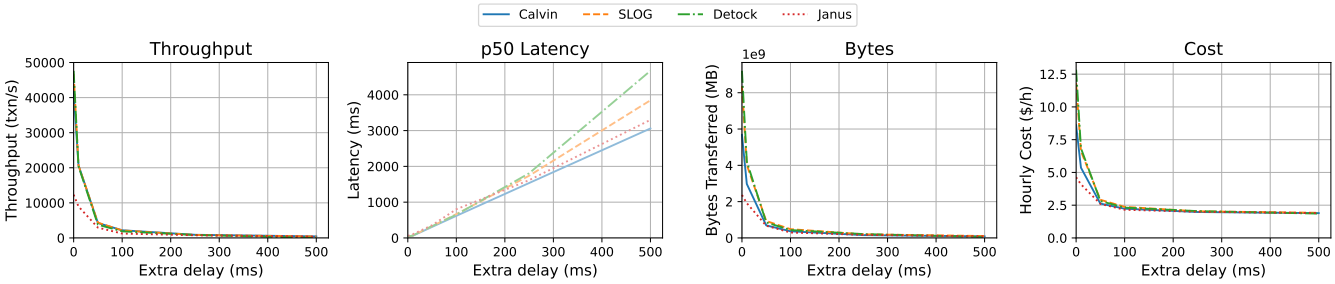

Figure 8: Results of the Network Delays Scenario.

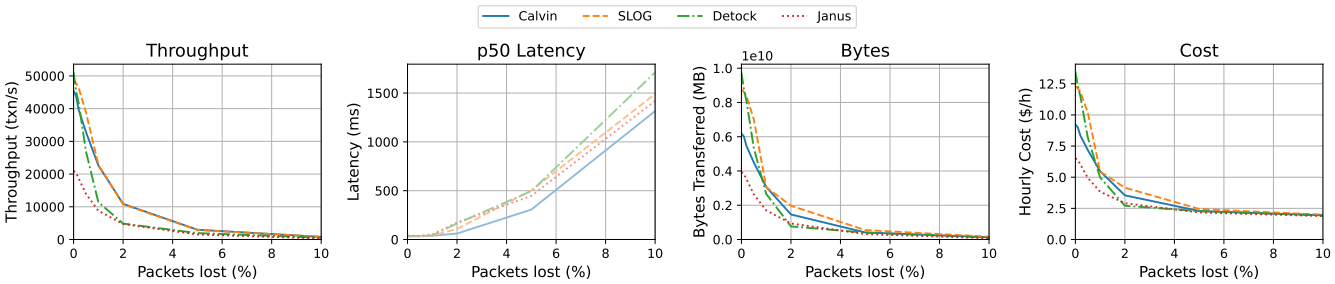

Figure 9: Results of the Packet Loss Scenario.

scales to roughly 1000 clients, then its throughput collapses to 0. The likely reason is that Janus's unified consensus and replication layer has to handle more and more dependency checks in the serialization graph, which overwhelms its CPUs and network links until it can no longer make progress.

4.2.5 Network Delays Scenario

This scenario explores how extra latency affects each system. We inject an additional delay on every cross-region link while keeping the packet loss at 0% and using the standard workload mix from Section 4.2.4. Figure 8 plots the results. The throughput decreases exponentially for all systems as the round-trip time grows. We note that the median latencies increase linearly with the injected delay, and Calvin's latency curve remains the lowest. This indicates that Calvin uses the smallest number of messages per transaction. Indeed, the coordinator only forwards the transactional input to the primary replica, which then broadcasts the messages in order.

4.2.6 Packet Loss Scenario

In this experiment, we inject random packet loss on the WAN links, ranging from 0% to 10%, while leaving the RTTs at their default values and the workload mix at its standard from Section 4.2.4. Once again, the throughput falls exponentially for all systems, and Calvin proves itself the most resilient in terms of latency. This again proves Calvin's lightweight communication, which minimizes the number and size of WAN messages that can be delayed or lost.

5 Responsible Research

This work does not involve personal data or any ethically sensitive information. We conducted all the experiments using synthetic data, which does not come from real users or real-world systems. Because of this, there are no ethical concerns related to privacy or security in our research.

We made efforts to ensure the reproducibility of our results. Section 4.1 describes the environment used for the experiments, along with the way we measured the performance and collected the metrics. The deployment is fully automated using Docker containers. The source code for the server, client, and experimental scripts will be available in a public repository[2], along with step-by-step instructions for setting up the benchmarking framework, running the experiments, and collecting the results. We understand that others who repeat our experiments may get slightly different results. This is expected in distributed systems research due to several factors, including the noise from background processes or the unpredictability of network communication. To reduce this variability, we ran the experiments several times and reported the average result. This helps in making our comparisons more reliable. In addition, we used fixed random seeds in our transaction generator, so that experiments with identical parameters follow the same behavior across different runs.

6 Discussion

Our experiments show that the PPS workload complements the commonly used workloads TPC-C and YCSB+T by covering several gaps they leave open, but it doesn't address every evaluation need. In this section, we outline the main advantages of using the PPS workload (Section 6.1) and also acknowledge its limitations (Section 6.2).

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

6.1 Advantages of the PPS Workload

Dependent Transactions. PPS models many-to-many relationships between products, parts, and suppliers. Thus, transactions need to work with *foreign key dependencies* when joining different tables, which is not the case in the single-table YCSB+T workload and the TPC-C warehouse hierarchy. In particular, the *OrderProduct* type needs to perform a read phase to discover all the relevant parts before being able to perform the actual updates. If another client changes the retrieved list of parts in between, then the transaction aborts and retries. Neither TPC-C nor YCSB+T ever forces the systems to abort, so PPS is the only one of these three workloads that stresses this case. We captured how increasing the skew factor impacts the resulting abort rates in Section 4.2.2.

Longer Transaction Footprint. PPS allows us to choose the number of parts per product, and thus control the size of the *read/write sets* of both *GetPartsByProduct* and *OrderProduct* transactions. In our experiments, we set this parameter to 10. The larger access sets in PPS particularly penalize the protocols whose cross-region coordination grows in complexity as the number of keys increases. Janus is an example of such a design, where a larger access set would lead to bigger messages and more potential conflicts, which pushes the transaction on the slow path that needs a second WAN round-trip. As a consequence, Janus's throughput relative to the other systems in our PPS baseline (Section 4.2.1) is far lower than in Nguyen et al.'s study [6], which used YCSB+T.

Fine-grained Home Region Control. Besides the ability to dynamically generate multi-home or single-home transactions, which is a feature already available in the Detock performance evaluation [6], our PPS-based benchmarking framework allows us to redirect a configurable fraction of transactions to a particular region. For a single-home transaction, we can select the exact region that will coordinate it, and for a multi-home transaction, we can pick a specific region to appear in the home set. This flexibility was essential for the sunflower experiment (Section 4.2.3), where we gradually shifted the traffic to a region to create a realistic *hotspot*.

6.2 Limitations of the PPS Workload

Limited Updates. Unfortunately, only two transactions in the PPS workload mix perform writes, namely *OrderProduct* and *UpdateProductPart*. Each of them modifies records from a *single table*. On the other hand, TPC-C offers a broader spectrum of write patterns, which can lead to more interesting contention challenges that PPS cannot reproduce.

Multi-home and Multi-partition Limitations. Because only the *OrderProduct* transaction can be configured to be multi-home or multi-partition, the workload mix must be adjusted to consider this. Even though we can choose the percentages of *OrderProduct* transactions in the mix, it is still a dependent transaction whose read-only first phase is always single-home and single-partition, and thus we cannot achieve an *overall combination* where 100% of the generated transactions are multi-home and multi-partition. On the other hand, even though YCSB+T is simpler, it allows arbitrary key distributions, so it can be used to reach any desired proportion of multi-home and multi-partition transactions.

7 Conclusions and Future Work

This work introduced a configurable benchmarking framework based on the PPS workload and demonstrated how it can be used to augment the standard workloads TPC-C and YCSB+T. Our benchmark adds dependent transactions, larger read/write sets, and fine-grained control over single-home and multi-home access patterns, which are key aspects when evaluating any geo-distributed database system.

Using this benchmark, we compared four such systems, Calvin, SLOG, Detock, and Janus, across six different scenarios. The results show clear tradeoffs. Calvin's global sequencer pays off once there's a majority of transactions that access multiple regions, whereas SLOG and Detock excel when most transactions can be executed locally. Janus's bottleneck is the required WAN round-trip approval, and the larger PPS access set can potentially lead to another WAN round-trip to solve the conflicts, which generally proved to have low performance in our two-region deployment. The skew experiment showed the hidden cost of having dependent transactions through increasing abort rates, while the sunflower experiment exposed how systems that are optimized based on locality can overload one replica.

Ultimately, the purpose of this paper is to raise awareness of the need for benchmarks suited for geo-distributed databases. Although PPS fills some gaps, it has its limitations. Future extensions could introduce additional transactions that are write-heavy or have the potential to span multiple regions. Moreover, it might be worth exploring how the results would be affected by having server-side reconnaissance (OLLP) for dependent transactions.

References

[1] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild *et al.*, "Spanner: Google's globally distributed database," *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 3, pp. 1–22, 2013.

[2] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li *et al.*, "{TAO}:{Facebook's} distributed data store for the social graph," in *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, 2013, pp. 49–60.

[3] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Highly available transactions: virtues and limitations (extended version)," *arXiv preprint arXiv:1302.0309*, 2013.

[4] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi, "Calvin: fast distributed transactions for partitioned database systems," in *Proceedings of the 2012 ACM SIGMOD international conference on management of data*, 2012, pp. 1–12.

[5] K. Ren, D. Li, and D. J. Abadi, "Slog: Serializable, low-latency, geo-replicated transactions," *Proceedings of the VLDB Endowment*, vol. 12, no. 11, 2019.

[6] C. D. Nguyen, J. K. Miller, and D. J. Abadi, "Detock: High performance multi-region transactions at scale,"

Proceedings of the ACM on Management of Data, vol. 1, no. 2, pp. 1–27, 2023.

[7] S. Mu, L. Nelson, W. Lloyd, and J. Li, "Consolidating concurrency control and consensus for commits under conflicts," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 517–532.

[8] S. T. Leutenegger and D. Dias, "A modeling study of the tpc-c benchmark," *ACM Sigmod Record*, vol. 22, no. 2, pp. 22–31, 1993.

[9] A. Dey, A. Fekete, R. Nambiar, and U. Röhm, "Ycsb+ t: Benchmarking web-scale transactional databases," in *2014 IEEE 30th International Conference on Data Engineering Workshops*. IEEE, 2014, pp. 223–230.

[10] L. Qu, Q. Wang, T. Chen, K. Li, R. Zhang, X. Zhou, Q. Xu, Z. Yang, C. Yang, W. Qian *et al.*, "Are current benchmarks adequate to evaluate distributed transactional databases?" *BenchCouncil Transactions on Benchmarks, Standards and Evaluations*, vol. 2, no. 1, p. 100031, 2022.

[11] R. Harding, D. Van Aken, A. Pavlo, and M. Stonebraker, "An evaluation of distributed concurrency control," *Proceedings of the VLDB Endowment*, vol. 10, no. 5, pp. 553–564, 2017.

[12] A. Thomson and D. J. Abadi, "The case for determinism in database systems," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 70–80, 2010.

[13] M. Serafini, R. Taft, A. J. Elmore, A. Pavlo, A. Aboulnaga, and M. Stonebraker, "Clay: Fine-grained adaptive partitioning for general database schemas," *Proceedings of the VLDB Endowment*, vol. 10, no. 4, pp. 445–456, 2016.

[14] T. P. P. Council, "Tpc benchmark c (standard specification, revision 5.11), 2010," *URL: http://www. tpc. org/tpcc*, p. 100, 2010.

A Use of Large Language Models

In compliance with the *Course Policy on the Use of Large Language Models (LLMs)*, LLMs were used exclusively for correcting the grammatical errors and improving the stylistic clarity, and not for generating ideas or drafting new text passages. The used prompts had the patterns: ”Please provide feedback on the readability of the following paragraph: ...” and ”Please fix the grammar in the following paragraph: ...”.