International Journal Software Engineering and Computer Science (IJSECS)
6 (2), 2026, 615-626
Published Online August 2026 in IJSECS (http://www.journal.lembagakita.com/index.php/ijsecs)
P-ISSN: 2776-4869, E-ISSN: 2776-3242. DOI: https://doi.org/10.35870/ijsecs.v6i2.7490.

RESEARCH ARTICLE Open Access

Performance and Stability Evaluation of Concurrency Models for I/O-Bound Services in Golang under High-Volume Transaction Load

Salsabila Yunita Sari [1*], **Felix David** [2].

[1*,2] Department of Informatics Engineering, Universitas Kristen Satya Wacana, Salatiga City, Central Java Province, Indonesia.

**Corresponding author*: 672022217@student.uksw.edu.

Received: June 10, 2026; Accepted: July 16, 2026; Published: August 1, 2026.

Abstract: Large-scale transaction processing in I/O-bound services can lead to performance bottlenecks and database instability, particularly on resource-constrained servers. An inappropriate concurrency strategy may cause excessive memory consumption and system failure under high workloads. This study compares the efficiency and stability of three concurrency models in Go: Sequential, Unlimited Goroutine, and Worker Pool with a Backpressure mechanism for processing 11 million transaction records against a PostgreSQL database. Testing was conducted in an isolated environment using Docker containers running on a Colima virtual machine configured with 4 CPU cores, 8 GB RAM, and a maximum of 50 database connections. Data were retrieved iteratively using Dynamic Keyset Filtering, while the batch size was calibrated through a preliminary tuning experiment. The results indicate that a batch size of 16,000 provided a balance between I/O efficiency and memory stability. The Unlimited Goroutine model failed to complete the workload due to an out-of-memory (OOM) condition. The Sequential model processed all records in 632 seconds with a throughput of 17,405 transactions per second (Tx/s). The 4-worker Worker Pool achieved the highest throughput at 21,917 Tx/s in 502 seconds, while the 8-worker Worker Pool achieved 19,755 Tx/s in 557 seconds with a 0% error rate and more consistent performance across the tested conditions. Based on the consistency of its performance, the 8-worker Worker Pool was selected as the preferred configuration in this study. These findings indicate that combining a Worker Pool with Backpressure and an appropriately calibrated batch size can improve the performance and stability of large-scale, Go-based transaction processing under the tested conditions.

Keywords: Concurrency; Golang; Worker Pool; Backpressure; I/O-Bound Workload.

1. Introduction

The development of modern backend systems has increased the need to process large volumes of data efficiently and reliably, particularly in scenarios commonly encountered in production environments such as large-scale transaction processing, data integration, and batch transaction updates in a database. In REST API-based architectures, the concurrency mechanism is an important factor affecting system throughput and server resource utilization, since the way concurrent work is scheduled and admitted into a system directly shapes how efficiently available hardware resources are used under load. The Go programming language addresses this need through concurrency primitives based on goroutines and channels, adopting the Communicating Sequential Processes (CSP) concept, which enables concurrent processes to communicate

through message passing rather than relying solely on shared memory and explicit locking (Malhotra, 2025). Unlike conventional operating-system threads, goroutines are managed by the Go runtime and have a relatively low initial memory overhead, allowing a large number of them to be created without immediately incurring the cost typically associated with OS-level threads (Nguyen, 2024). Building on this capability, this study focuses on a channel-based Worker Pool with Backpressure because it provides explicit control over the number of concurrent workers and the amount of work entering the processing queue, in contrast to simply allowing goroutines to be created without bound. A related mechanism sometimes considered for controlling execution parallelism in Go is the GOMAXPROCS parameter, which determines the number of logical processors available for simultaneous execution of Go code but does not directly limit the number of goroutines created by an application or the rate at which new work enters the system. Because many goroutines may remain in a waiting state while database operations are in progress in I/O-bound workloads such as database processing, constraining GOMAXPROCS alone does not necessarily prevent the accumulation of goroutines or contention for database resources. A Worker Pool with Backpressure, by contrast, explicitly limits the number of active workers and controls the amount of pending work through a buffered channel, and is therefore expected to provide better control over memory usage and workload admission than relying on GOMAXPROCS alone.

The ease of creating goroutines can nevertheless introduce resource-management challenges in I/O-bound services, where a substantial portion of the workload involves network or database operations and goroutines may frequently remain in a waiting state while awaiting database responses. When the number of concurrently active goroutines is not properly controlled, memory consumption and database connection contention may increase, while additional scheduling overhead can affect overall service stability (Kennedy, 2018). This concern is particularly relevant because many services in production environments operate on servers with limited computational resources, where an inappropriate concurrency strategy may result in increased CPU consumption, higher memory usage, and reduced stability during periods of high load. Previous work has reported that unbounded goroutine creation can produce substantial memory pressure and increase garbage-collection overhead in high-throughput services, reinforcing the need for a concurrency strategy that keeps resource consumption within predictable limits (Sharma, 2025).

The worker pool pattern offers such a strategy by maintaining a fixed number of workers that retrieve tasks from a shared queue, which allows the number of concurrently executing goroutines to be controlled more explicitly and can help regulate resource consumption under high workloads (Doan, 2025). Complementing this pattern, backpressure can be used to control the rate at which new tasks enter the processing pipeline when the available workers are already occupied, preventing unbounded accumulation of pending work in memory (Matteussi et al., 2022). The case study in this research represents a large-scale batch transaction processing scenario in which each transaction is processed to calculate accumulated progress per member, update the evaluation status in the transaction table, and perform batch updates to the progress table. These operations involve direct interaction with a PostgreSQL database, including iterative data retrieval using cursor-based pagination and writing calculation results through bulk upsert operations, and the resulting workload is predominantly I/O-bound because the processing involves repeated database read and write operations rather than intensive computation.

To maintain experimental control, this study uses a synthetic dataset designed to reproduce the scale and structural characteristics of the case study, so that the actual production transaction records are not directly used in the experiment. Instead, the workload represents approximately 11,000,000 primary transaction records and nearly 33 million transaction-detail records, based on the observed scale of the operational workload, and corresponds to approximately 24 hours of accumulated transaction activity with an average workload of approximately 127 Transactions Per Second (TPS). This figure is used as the baseline workload for evaluating how quickly the system can process a large accumulated transaction queue, although in actual operational environments transaction volumes may vary over time and can increase during peak periods; the workload used in this study should therefore be interpreted as a production-scale synthetic workload rather than a direct reproduction of the production transaction records.

Based on the studies reviewed in this research, limited empirical evidence is available on comparative evaluation of concurrency models across normal and overload conditions for I/O-bound workloads at a scale of millions of transactions involving actual database operations. This study therefore aims to evaluate and compare the efficiency and stability of three concurrency models—Sequential, Unlimited Goroutine, and Worker Pool with Backpressure—in processing an 11-million-record I/O-bound workload on a resource-constrained Go-based backend service, with the intention of providing empirical evidence on the performance and stability characteristics of these concurrency models under different load conditions.

2. Related Work

2.1 Concurrency in the Go Language

Go uses goroutines as lightweight units of execution managed by the Go runtime rather than directly by the operating system (Nguyen, 2024). Compared with conventional operating-system threads, goroutines have a relatively small initial stack size, which can grow automatically when additional memory is required. This design allows Go applications to manage large numbers of concurrent goroutines with relatively low initial memory overhead (Nguyen, 2024). Goroutines communicate using channels, which are based on the Communicating Sequential Processes (CSP) concept (Malhotra, 2025). Channels can be buffered, allowing a limited number of values to be queued, or unbuffered, requiring synchronization between the sender and receiver. In this study, a buffered channel is used in the Worker Pool model as both a task queue and a backpressure mechanism. When the channel reaches its capacity, the producer blocks until workers consume available tasks.

2.2 The Go Scheduler Model (GMP)

Behind the goroutine abstraction, Go uses the GMP scheduling model (Ahmad, 2026). Its three main components are G (Goroutine), which represents the unit of execution; M (Machine), which represents an operating-system thread; and P (Processor), which provides the logical context required for a goroutine to execute on an M. The number of logical processors available for simultaneous execution of Go code is controlled by GOMAXPROCS (Ahmad, 2026). When a goroutine is waiting for a database response or another I/O operation, the scheduler can allow the associated operating-system thread to execute another runnable goroutine. This mechanism enables other goroutines to continue executing while an I/O-bound operation is waiting. However, concurrency can still become inefficient when a large number of goroutines compete for shared resources such as database connections, potentially resulting in queue build-up and reduced overall system performance (Kennedy, 2018).

2.3 Three Concurrency Models Tested

The Sequential model is the simplest approach. Data is retrieved in batches and processed sequentially within a single main goroutine. After the data in a batch has been processed, the results are sent to the database. Its main advantage is its simple execution flow and the absence of synchronization requirements between concurrent workers. However, CPU resources may remain underutilized while the application is waiting for database I/O operations (Kennedy, 2018). The Unlimited Goroutine model attempts to increase concurrency by creating a new goroutine for each data item in a batch. These goroutines execute concurrently, and their results are merged before being written to the database. Because multiple goroutines access shared structures such as progressMap and trxIDs, synchronization using sync.Mutex is required to prevent race conditions and data corruption. When a large batch contains thousands of data items, creating a corresponding number of goroutines can substantially increase memory consumption and scheduling overhead. In database-intensive workloads, uncontrolled concurrency can therefore increase resource pressure and affect system stability (Sharma, 2025). The Worker Pool with Backpressure model uses a fixed number of goroutines, referred to as workers, which are created at the beginning of processing and remain active until all data has been processed. Data retrieved from the database is placed into taskChan, a buffered channel. Each worker retrieves tasks from the channel and stores intermediate results in local variables such as localProgress and localTrxIDs. Therefore, a mutex is not required for these worker-local result variables. A worker sends data to the database through a flush operation when its local record reaches flushThreshold or when all data has been processed. Because the channel has a limited capacity, the producer blocks when the channel is full, preventing unlimited accumulation of pending tasks and providing a form of backpressure (Doan, 2025; Matteussi *et al.*, 2022).

2.4 Related Work and the Position of This Study

Previous studies have examined concurrency in Go from several perspectives. Malhotra (2025) discussed worker pool and fan-in/fan-out approaches for concurrent backend processing, while Nguyen (2024) discussed the memory characteristics of goroutines and the implications of creating large numbers of concurrent goroutines. Dilley and Lange (2019) investigated concurrency practices in Go projects and reported the use of channel-based communication in real-world Go software, supporting the relevance of channels for coordinating concurrent tasks. Sharma (2025) and Doan (2025) discussed concurrency and worker pool considerations in high-volume and database-intensive Go services. Outside the Go ecosystem, Reza and Supatmi (2023) evaluated concurrency performance in a relational database management system within a virtualized environment. Ling *et al.* (2000) examined thread pool sizing from a queueing-theory perspective, which is relevant to the analysis of worker-count selection in concurrent systems. Based on the studies reviewed in this research, the present study differs in its combination of a large-scale synthetic transaction workload, actual

PostgreSQL database operations, and explicit evaluation under two load phases: normal operation and overload. The workload consists of approximately 11 million primary transaction records and is designed to represent the scale of a large operational transaction-processing scenario. Therefore, the study focuses not only on processing performance but also on system stability under different load conditions. Table 1 summarizes the comparison between this study and the reviewed related work.

Table 1. Comparison of Related Work

Source	Research Focus	Language/Platform	Workload Type	Data Scale	Real DB Operations	Stability Analysis
Malhotra (2025)	Worker Pool, fan-in/fan-out	Go	Mixed (I/O-bound + CPU)	Medium	No	Partial
Nguyen (2024)	Goroutine memory behavior	Go	General	Small	No	No
Ahmad (2026)	GMP Scheduler mechanism	Go Runtime	I/O-bound	Conceptual	No	No
Kennedy (2018)	Go Scheduler, I/O concurrency	Go	I/O-bound	Medium	No	Partial
Sharma (2025)	High-volume DB change streams	Go + PostgreSQL	I/O-bound	Large	Yes	Partial
Doan (2025)	Worker pool: production lessons	Go	I/O-bound	Large	Yes	Yes
Surwase (2024)	Worker pool, batch processing	Go	I/O-bound	Medium	Yes	Partial
Reza & Supatmi (2023)	RDBMS concurrency performance in VM	MySQL/VM	Mixed	Medium	Yes	Partial
This study	Comparison of three concurrency models	Go + PostgreSQL	I/O-bound	11 million records	Yes	Yes

The comparison indicates that the reviewed studies generally address specific aspects of concurrency, such as goroutine behavior, scheduler mechanisms, worker pool design, or database concurrency. However, fewer studies combine a large-scale transaction workload, actual database operations, and explicit stability evaluation across both normal and overload conditions. This combination forms the empirical focus of the present study.

3. Methodology

3.1 Research Design

This study is a comparative experimental study in which three concurrency models implemented in the Go programming language—Sequential, Unlimited Goroutine, and Worker Pool with Backpressure—are evaluated under controlled experimental conditions. The models are tested using the same dataset, hardware configuration, and database configuration so that differences in throughput, resource usage, and error rate can be primarily attributed to the concurrency strategy.

3.2 Testing Environment

All tests were conducted in an isolated local environment to minimize the influence of external network conditions. The host device was a laptop with an 8-core CPU and 16 GB of unified memory. To reduce the direct influence of the host device specification, the backend service and database were executed inside a Linux virtual machine (VM) using Colima. The main testing environment was configured with 4 CPU cores and 8 GB of RAM. The database used in the experiment was PostgreSQL, running inside a Docker container. The maximum number of database connections was configured as max_connections = 50, which was lower than the PostgreSQL default of 100 connections. This configuration was used to limit database resource

consumption and maintain a controlled environment under the constrained VM configuration. The software environment consisted of Go 1.26.0 (darwin/arm64 on the host), PostgreSQL 17.9, Docker 29.2.1, and Colima 0.10.1. [cek data: confirm whether PostgreSQL 17.9 or the postgres:15 image was actually used.] The PostgreSQL memory parameters, including shared_buffers and work_mem, were configured using the default values of the PostgreSQL image used in the experiment. The exact image version and parameter values should be reported consistently to support reproducibility.

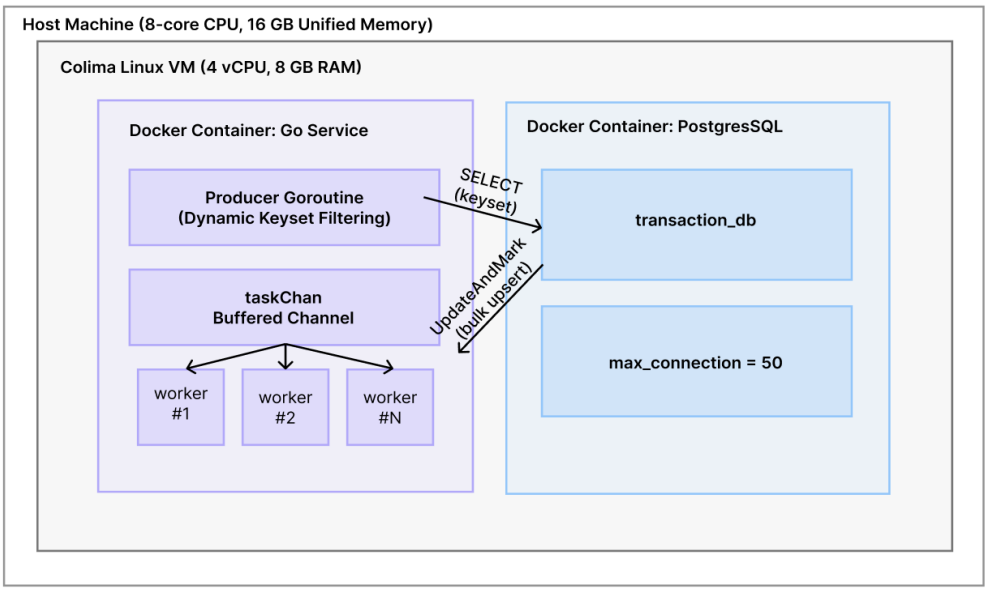

Figure 1. Deployment Architecture and Data Flow of the Experimental Environment

Figure 1 illustrates the deployment architecture used in the experiment. The Go service and PostgreSQL database were executed as two separate Docker containers within the same Colima VM and communicated through the internal container network.

3.3 Dataset and Processing Scenario

The dataset used in the experiment consisted of 11,000,000 synthetic transaction records stored in PostgreSQL. Each processing cycle consisted of three main stages: (1) retrieving data from the database using Dynamic Keyset Filtering to avoid processing the same transaction more than once; (2) calculating accumulated progress for each member based on the retrieved data; and (3) writing the processing results back to the database by updating the evaluation status and progress amount through the UpdateAndMark function. Because the workload primarily consists of database read and write operations, it is classified as I/O-bound. Performance is therefore expected to be strongly influenced by database I/O and response latency rather than solely by CPU computation. This workload provides a basis for evaluating how different concurrency strategies affect processing speed and system resource usage.

Table 2. Experimental Data Specification Characteristics

Table Name	Number of Records
Users	100,000
Transaction	11,000,000
Transaction_detail (maximum 5 per transaction)	32,995,065

The experiment used relational data with a one-to-many schema generated using synthetic data generation. The relationships between entities were generated according to the defined schema, with a maximum of five transaction-detail records associated with each transaction. The volume and role of each table are summarized in Table 2.

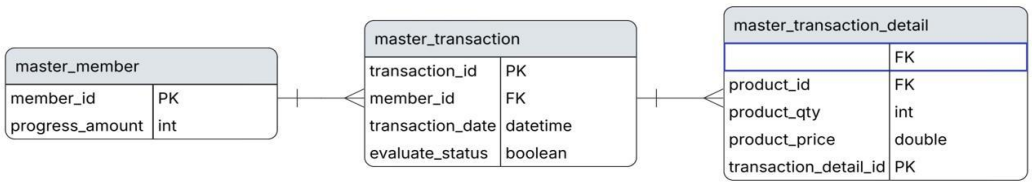

Figure 2. Relational Database Schema Model

3.4 System Architecture and Worker Pool Mechanism

Figure 3 illustrates the data flow and internal control of the Worker Pool with Backpressure model. The Producer goroutine retrieves rows from PostgreSQL and sends them to taskChan, a buffered channel with a fixed capacity. A fixed number of worker goroutines retrieve tasks from taskChan, accumulate processing results in worker-local buffers, and flush the results to the database when the local buffer reaches flushThreshold. When taskChan reaches its capacity, the Producer blocks until one or more workers consume available tasks. This blocking behavior forms the backpressure mechanism and prevents unlimited accumulation of pending tasks in memory.

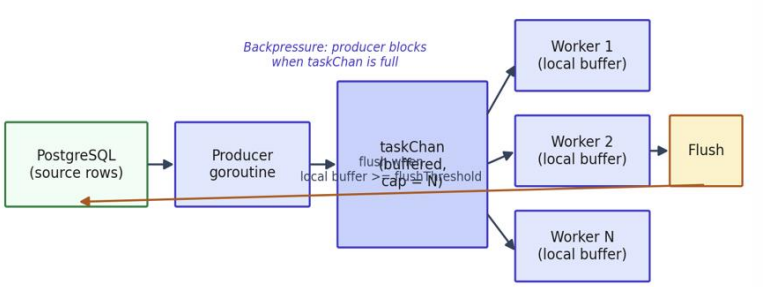

Figure 3. Worker Pool with Backpressure: Data and Control Flow

The pseudocode for the Worker Pool with Backpressure mechanism is as follows:

```
function Producer(db, taskChan):
    lastID = 0
    While true:
        rows = db.Query("SELECT * FROM transaction WHERE id > ? ORDER BY id LIMIT ?", lastID, batchSize)
        if rows is empty: break
        for row in rows:
taskChan <- row      // blocks if taskChan is full  -> BACKPRESSURE
        lastID = rows.last().id
    close(taskChan)

function Worker(id, taskChan, db):
    localProgress = {}
    localTrxIDs = []
    for task in taskChan:
        localProgress[task.memberID] += compute(task)
        localTrxIDs.append(task.id)
        if len(localTrxIDs) >= flushThreshold:
db.UpdateAndMark(localProgress, localTrxIDs)   // flush
            localProgress = {}
            localTrxIDs = []
    if len(localTrxIDs) > 0:
db.UpdateAndMark(localProgress, localTrxIDs)       // final flush

function Main():
    taskChan = make(channel, capacity = N)
    go Producer(db, taskChan)
    for i in 1..numWorkers:
        go Worker(i, taskChan, db)
    WaitAll()
```

3.5 Development Process

The first model, Sequential, retrieves data in batches and processes each transaction sequentially within a for loop before calling the UpdateAndMark function. No additional goroutines or mutex synchronization are required for the processing logic. This model serves as the baseline for evaluating the performance effects of concurrency. In the second model, Unlimited Goroutine, each transaction retrieved in a batch is processed by a separate goroutine. Because multiple goroutines access shared variables such as evaluate_status and trxIDs, sync.Mutex is used to prevent concurrent access from causing data corruption. All goroutines are synchronized using sync.WaitGroup before the results are sent to the database. With batchSize = 16,000, up to 16,000 goroutines may be created during a single batch. Such unbounded concurrency can increase memory consumption and scheduling overhead when processing millions of records. The third model, Worker Pool with Backpressure, consists of three main components. First, the Worker Pool creates a fixed number of goroutines at the beginning of processing, and the workers remain active until all tasks have been consumed. Each worker maintains local variables such as localProgress and localTrxIDs, reducing the need for mutex synchronization on these variables. Second, the Producer retrieves data from the database and places the tasks into taskChan, a buffered channel. The Producer blocks when the channel reaches its capacity, providing the backpressure

mechanism. Third, workers perform local flush operations when the number of accumulated records reaches flushThreshold and perform a final flush after all tasks have been processed.

3.6 Experimental Setup

The experiment was designed to evaluate the efficiency and stability of the three concurrency models using the following parameters.

1) Parameter calibration**.**
   The tested batch sizes were 4,000, 8,000, 16,000, and 32,000 records per iteration. These values were selected using a successive-doubling sequence to provide progressively larger workload configurations and to observe the effect of increasing batch size on system performance and resource consumption.
2) Independent experimental factor**.**
   The main experimental factor was the concurrency model, consisting of three approaches: Sequential, Unlimited Goroutine, and Worker Pool with 4, 8, and 16 workers. The different worker counts in the Worker Pool model were evaluated to determine how the number of concurrent workers affected system performance and stability.
3) Dependent variables**.**
   The dependent variables were CPU utilization (%), RAM usage (MB), error rate (%), and throughput (records per second). Throughput was calculated using Formula 1:

$$T = \frac{N}{t} \quad (1)$$

Where:
T = Throughput in records per second;
N = Total number of processed records;
t = Total execution time in seconds.

CPU utilization and RAM usage were recorded during the execution of each test to evaluate the resource consumption of each concurrency model. The error rate was calculated based on the number of failed processing operations relative to the total number of records processed. Throughput was calculated from the total number of successfully processed records divided by the total execution time. Before each testing session, the transaction data was reset to its initial state by restoring the evaluate_status and progress_amount values to their default values. This procedure ensured that each model was tested using an equivalent dataset state. The Sequential model was used as the baseline for comparing throughput across the tested concurrency models. Each experimental configuration was executed under the same testing conditions, including the dataset, VM configuration, database configuration, and processing procedure. The results were then recorded for comparison based on throughput, CPU utilization, RAM usage, and error rate.

4. Result and Discussion

4.1 Results

4.1.1 Batch Size Tuning

Before conducting the large-scale comparative experiment using 11,000,000 transaction records, a batch-size tuning experiment was conducted using approximately 1,000,000 records. The purpose of this experiment was to identify a batch size that could provide a reasonable balance between database I/O efficiency and memory usage during data processing. The Sequential model was used for this calibration, with batch sizes of 4,000, 8,000, 16,000, and 32,000 records. The results of the batch-size tuning experiment are presented in Table 3.

Table 3. Batch Size Tuning Test Results

Batch Size	Total Test Data	Number of Cycles	Start Time	End Time	Total Time	Average Throughput	Final Status
4,000	1,000,000	250	09:41:38	09:42:41	63 s	15,873 records/s	Success
8,000	1,000,000	125	09:43:40	09:44:37	57 s	17,544 records/s	Success
16,000	1,008,000	63	09:45:38	09:46:35	57 s	17,684 records/s	Success
32,000	1,024,000	32	09:48:06	09:49:03	57 s	17,965 records/s	Success

The results show that throughput increased as the batch size increased. A batch size of 4,000 records produced the lowest throughput at 15,873 records/s with a total processing time of 63 seconds. Increasing the batch size to 8,000 records increased throughput to 17,544 records/s, while batch sizes of 16,000 and 32,000 records

achieved 17,684 and 17,965 records/s, respectively. Although the 32,000-record batch produced the highest measured throughput, the difference from the 16,000-record batch was relatively small. The throughput increased by 281 records/s, or approximately 1.6%, when the batch size was doubled from 16,000 to 32,000 records. Similarly, increasing the batch size from 8,000 to 16,000 records resulted in an increase of 140 records/s, or approximately 0.8%. The experiment logs also showed differences in per-cycle processing duration. The first-cycle durations for batch sizes of 4,000 and 8,000 were 255.74 ms and 478.81 ms, respectively. For batch sizes of 16,000 and 32,000, the first-cycle durations were approximately 1.35 seconds and 2.00 seconds, respectively. Subsequent cycles showed shorter processing times, indicating that the initial processing cycle required additional initialization and resource allocation. Based on the overall results, the 16,000-record batch size was selected as the reference batch size for the subsequent concurrency experiments. Although the 32,000-record configuration achieved slightly higher throughput, the performance gain was relatively small compared with the increase in the amount of data processed in each cycle. The 16,000-record configuration therefore provided a practical balance between processing efficiency and batch size.

4.1.2 Concurrency Model Implementation Results

The three concurrency strategies were subsequently evaluated using a batch size of 16,000 records on a VM configured with 4 CPU cores and 8 GB RAM. PostgreSQL was configured with a maximum of 50 database connections. The experiment evaluated the Sequential model, Unlimited Goroutine model, and Worker Pool with 4, 8, and 16 workers. The throughput, execution time, error rate, and final status are presented in Table 4.

Table 4. Concurrency Model Test Results: Throughput and Execution Time

Concurrency Model	Batch Size	Workers	Total Execution Time	Throughput (Tx/s)	Error Rate	Final System Status
Sequential	16,000	–	632 s	17,405	0%	Completed successfully
Unlimited Goroutine	16,000	–	–	–	100%	Failed (Database OOM)
Worker Pool + Backpressure	16,000	4	502 s	21,917	0%	Completed successfully
Worker Pool + Backpressure	16,000	8	557 s	19,755	0%	Completed successfully
Worker Pool + Backpressure	16,000	16	624 s	17,620	0%	Resource contention

The Sequential model completed the processing of 11,000,000 records in 632 seconds, achieving a throughput of 17,405 Tx/s with an error rate of 0%. The Unlimited Goroutine model failed during execution and recorded an error rate of 100%. The test was terminated after the database encountered an out-of-memory condition. The Worker Pool with 4 workers achieved the highest throughput among the successful configurations, processing the dataset in 502 seconds at 21,917 Tx/s. The Worker Pool with 8 workers completed the processing in 557 seconds with a throughput of 19,755 Tx/s. The 16-worker configuration completed the processing in 624 seconds with a throughput of 17,620 Tx/s, which was only slightly higher than the Sequential baseline. Resource utilization results are presented in Table 5.

Table 5. Concurrency Model Test Results: Resource Usage

Concurrency Model	Workers	CPU Usage (%)	RAM Usage (MiB)
Sequential	–	89.7% peak	617 peak
Unlimited Goroutine	–	–	–
Worker Pool + Backpressure	4	96.57 → 191.45 → 146.43	175.3 → 300.4 → 345.8
Worker Pool + Backpressure	8	95.65 → 198.33 → 204.67	183.4 → 345.3 → 398.1
Worker Pool + Backpressure	16	398.87% peak	790 peak

The 4-worker configuration recorded CPU utilization of 96.57% at the beginning, 191.45% in the middle, and 146.43% at the end of processing. RAM usage increased from 175.3 MiB to 345.8 MiB. The 8-worker configuration recorded CPU utilization of 95.65%, 198.33%, and 204.67% at the beginning, middle, and end of processing, respectively. RAM usage increased from 183.4 MiB to 398.1 MiB. The 16-worker configuration recorded a peak CPU utilization of 398.87% and peak RAM usage of 790 MiB.

4.1.3 Testing on the Default VM

Additional testing was conducted using a VM configured with 2 CPU cores and 2 GB RAM. The Sequential model and Worker Pool with 4 workers were tested using the same processing procedure. The results are presented in Table 6.

Table 6. Additional Test Results on Default and Optimized VM Configurations

Concurrency Model	VM Configuration	Total Time	Throughput	Difference from Optimized VM
Sequential	Default (2 cores, 2 GB)	683s	16,100 Tx/s	51 s slower
Sequential	Optimized (4 cores, 8 GB)	632s	17,405 Tx/s	Baseline
Worker Pool (4 workers)	Default (2 cores, 2 GB)	474s	23,164 Tx/s	28 s faster; +1,247 Tx/s
Worker Pool (4 workers)	Optimized (4 cores, 8 GB)	502s	21,917 Tx/s	Baseline

On the 2-core, 2-GB VM, the Sequential model completed the processing in 683 seconds with a throughput of 16,100 Tx/s. The Worker Pool with 4 workers completed the same workload in 474 seconds with a throughput of 23,164 Tx/s.

4.1.4 Cold-Start Testing

Additional testing was conducted by restarting the database container before execution. This procedure was intended to examine the effect of the PostgreSQL cold-start condition on processing performance. The results indicated that the Worker Pool configuration experienced a throughput reduction of approximately 24–31% under cold-start conditions, while the Sequential model experienced a reduction of approximately 3.9%. These results indicate that the concurrency models did not respond identically to changes in the database initial state.

4.2 Discussion

The batch-size experiment shows that increasing the batch size improved throughput, particularly when the batch size increased from 4,000 to 8,000 records. However, the improvement became much smaller at larger batch sizes. Increasing the batch size from 16,000 to 32,000 records increased throughput from 17,684 to 17,965 records/s, representing only about 1.6% improvement. Although the 32,000-record configuration achieved the highest throughput, the gain was relatively small compared with the increase in data processed per cycle. For this reason, the 16,000-record configuration was selected as the reference batch size because it provided nearly the same throughput while requiring a smaller workload per processing cycle. The difference in the total records processed during the tuning stage—1,000,000 records for the 4,000- and 8,000-record configurations, 1,008,000 records for the 16,000-record configuration, and 1,024,000 records for the 32,000-record configuration—reflects the use of complete processing cycles based on the selected batch size. The longer processing time observed in the initial and final cycles of larger batches may be related to memory allocation and runtime behavior, although confirming this relationship would require additional profiling of heap allocation and garbage collection activity.

The comparison of the concurrency models demonstrates that controlled concurrency produced higher throughput than sequential processing, whereas unrestricted goroutine creation was unable to complete the workload. The Sequential model processed all 11,000,000 records in 632 seconds with a throughput of 17,405 Tx/s and served as the baseline. The Worker Pool with 4 workers achieved the highest throughput at 21,917 Tx/s, representing approximately a 25.9% improvement over the Sequential model. Increasing the number of workers did not produce proportional performance gains, as the 8-worker configuration achieved 19,755 Tx/s and the 16-worker configuration achieved 17,620 Tx/s. These results indicate that increasing the number of concurrent workers beyond the tested level of four did not improve throughput under the experimental conditions. The Unlimited Goroutine model failed because of a database OOM condition, demonstrating that unrestricted concurrency was not suitable for the large-scale I/O-bound workload evaluated in this study. The reduced performance of the 16-worker configuration is more appropriately interpreted as increased concurrency overhead and resource contention rather than being attributed to a single specific factor.

The resource utilization results reinforce this pattern. RAM usage increased from 345.8 MiB with 4 workers to 398.1 MiB with 8 workers and reached 790 MiB with 16 workers. The 16-worker configuration also recorded the highest CPU utilization at 398.87%. Because the testing environment used four CPU cores, this value represents aggregate CPU utilization across the available cores rather than utilization of a single core. Despite consuming substantially more CPU and memory resources, the 16-worker configuration did not outperform the 4-worker configuration. These findings indicate that worker count should be matched to the available processing resources and workload characteristics rather than increased indiscriminately.

The results also demonstrate the practical benefit of the backpressure mechanism. Unlike the Unlimited Goroutine model, all Worker Pool configurations completed the workload successfully because the bounded taskChan limited the number of pending tasks by blocking the producer when the channel reached its capacity.

This prevented uncontrolled accumulation of work and helped maintain processing stability throughout execution. The findings therefore support the use of bounded concurrency for large-scale I/O-bound workloads, although the results should be interpreted within the tested dataset, PostgreSQL configuration, VM resources, batch size, and worker configurations.

An additional experiment using a smaller VM configuration produced an interesting observation. The Worker Pool with 4 workers completed the workload in 474 seconds on the 2-core, 2-GB VM, compared with 502 seconds on the 4-core, 8-GB VM, whereas the Sequential model became slower on the smaller VM, requiring 683 seconds instead of 632 seconds. This indicates that increasing CPU cores and memory capacity did not automatically translate into higher throughput for the Worker Pool configuration under the tested workload. The difference is likely influenced by the interaction between concurrency, database I/O, scheduling behavior, and VM resource allocation. However, the available measurements do not identify the exact contribution of each factor, so this finding should be interpreted as an observed characteristic of the tested environment rather than a general performance rule.

The cold-start experiment further shows that the Worker Pool configuration was more sensitive to the initial database condition than the Sequential model. After restarting the database container, the Worker Pool experienced a throughput reduction of approximately 24–31%, whereas the Sequential model decreased by approximately 3.9%. These results suggest that concurrent processing relied more heavily on the initial state of the database cache than sequential processing. Nevertheless, the available data do not indicate that PostgreSQL shared buffers alone were responsible for the observed difference, as other initialization factors may also have contributed. Future experiments would benefit from repeated trials with a consistent database restart procedure to provide stronger evidence regarding the effect of warm-cache and cold-start conditions.

The findings indicate that controlled concurrency provided a better balance between throughput and resource consumption than either sequential processing or unrestricted goroutine creation for the tested I/O-bound workload. The Worker Pool with 4 workers achieved the highest measured throughput while maintaining substantially lower resource consumption than the 16-worker configuration, whereas the Unlimited Goroutine model failed because of a database OOM condition. Under the experimental conditions of this study, the Worker Pool with 4 workers can therefore be considered the most effective tested configuration in terms of throughput and resource efficiency. However, this finding should not be interpreted as a universal recommendation because the appropriate worker count may vary depending on workload characteristics, database capacity, batch size, available CPU and memory resources, and the underlying application architecture.

5. Conclusion and Recommendations

This study evaluated three Go concurrency models—Sequential, Unlimited Goroutine, and Worker Pool with Backpressure—for processing 11,000,000 transaction records. The batch-size experiment showed that throughput increased with larger batches but the gain became marginal beyond 16,000 records. Therefore, a batch size of 16,000 was selected as the reference configuration because it provided high throughput without requiring the largest batch capacity. The concurrency results showed that the Worker Pool provided better performance than Sequential processing, while unrestricted goroutine creation failed because of a database out-of-memory (OOM) condition. The 4-worker Worker Pool achieved the highest throughput at 21,917 Tx/s in 502 seconds, compared with 17,405 Tx/s for Sequential processing. The 8-worker and 16-worker configurations achieved 19,755 Tx/s and 17,620 Tx/s, respectively, with higher worker counts consuming more resources without improving throughput. Therefore, under the tested conditions, the 4-worker Worker Pool with Backpressure was the best-performing configuration. The results do not support describing the 8-worker configuration as optimal unless additional stability measurements are provided.

Testing on the smaller 2-core, 2-GB VM produced an unexpected result: the 4-worker Worker Pool achieved 23,164 Tx/s, higher than the 21,917 Tx/s obtained on the 4-core, 8-GB VM. This finding should be treated as an observation rather than evidence that a smaller VM is inherently more efficient, because the available measurements do not identify its specific cause. The study is limited by the use of synthetic data, a Colima-based virtualized environment, and differences in PostgreSQL initial conditions between some tests. These factors may limit the generalization of the findings to other production environments. Future research should repeat the experiments on physical hardware, use controlled cold-start procedures with multiple repetitions, evaluate worker counts across different hardware configurations, examine the effect of database connection-pool size, and apply pprof to measure heap allocation and garbage-collection behavior directly.

References

Ahmad, A. (2026). *Understanding goroutines & the Go scheduler (GMP model).* https://blog.arishahmad.in/understanding-goroutines-the-go-scheduler-gmp-model

Dilley, N., & Lange, J. (2019). An empirical study of messaging passing concurrency in Go projects. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (pp. 377–387). IEEE. https://doi.org/10.1109/SANER.2019.8668036

Dinh-Tuan, H. (2025). Unikernels vs. containers: A runtime-level performance comparison for resource-constrained edge workloads. In *2025 8th Conference on Cloud and Internet of Things (CIoT)*. IEEE.

Doan, S. (2025, May). *How worker pools saved our Go services: Lessons from the trenches.* Medium. https://medium.com/@sanhdoan/how-worker-pools-saved-out-go-services-lessons-from-the-trenches-e55167a6b353

Hesse, G., Matthies, C., & Uflacker, M. (2020). How fast can we insert? An empirical performance evaluation of Apache Kafka. In *2020 IEEE International Conference on Parallel and Distributed Systems (ICPADS)* (pp. 641–648). IEEE.

Kar, S., Rehrmann, R., Mukhopadhyay, A., Alt, B., Ciucu, F., Koeppl, H., Binnig, C., & Rizk, A. (2021). On the throughput optimization in large-scale batch-processing systems. *ACM SIGMETRICS Performance Evaluation Review, 48*(3), 128–129. https://doi.org/10.1145/3453953.3453982

Kennedy, B. (2018, December). *Scheduling in Go: Part III – Concurrency.* Ardan Labs. https://www.ardanlabs.com/blog/2018/12/scheduling-in-go-part3.html

Le Noac'h, P., Costan, A., & Bougé, L. (2017). A performance evaluation of Apache Kafka in support of big data streaming applications. In *2017 IEEE International Conference on Big Data*. IEEE.

Ling, Y., Mullen, T., & Lin, X. (2000). Analysis of optimal thread pool size. *ACM SIGOPS Operating Systems Review, 34*(2), 42–55. https://doi.org/10.1145/346152.346320

Malhotra, A. (2025). Concurrency patterns in Golang: Real-world use cases and performance analysis. *Journal of Computing and Software Technologies, 5*(1), 1–14. https://al-kindipublishers.org/index.php/jcsts/article/view/10083

Matteussi, K. J., dos Anjos, J. C. S., Leithardt, V. R. Q., & Geyer, C. F. R. (2022). Performance evaluation analysis of Spark Streaming backpressure for data-intensive pipelines. *Sensors, 22*(13), 4756. https://doi.org/10.3390/s22134756

Nguyen, T. (2024, July). *A comprehensive guide to concurrency in Golang.* Relia Software. https://reliasoftware.com/blog/concurrency-in-golang

Reza, D., & Supatmi, S. (2023). Concurrency performance analysis on MySQL RDBMS in virtual machine environment. In *2023 9th International Conference on Science and Technology (ICSPIS)* (pp. 1–7). IEEE.

Sharma, P. (2025, December). *Designing high throughput Go services for continuous database change streams.* DEV Community. https://dev.to/sharmaprash/golang-optimizations-for-high-volume-services-dij

Surwase, R. K. (2024, April). *Efficient concurrency in Go: A deep dive into the worker pool pattern for batch processing.* Medium. https://rksurwase.medium.com/efficient-concurrency-in-go-a-deep-dive-into-the-worker-pool-pattern-for-batch-processing-73cac5a5bdca

Wu, M., *et al.* (2020). Platinum: A CPU-efficient concurrent garbage collector for tail-reduction of interactive services. In *2020 USENIX Annual Technical Conference (USENIX ATC)* (pp. 159–172).

Yıldırım, H. (2024, February). *CPU vs I/O bound benchmarking in Go.* Medium. https://medium.com/@halilylm/cpu-vs-i-o-bound-benchmarbing-in-go-fcd4f053694e

Zhao, J., Zhou, X., Chang, S.-Y., & Xu, C. (2023). Let it go: Relieving garbage collection pain for latency critical applications in Golang. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing (HPDC '23)* (pp. 169–180).