

Seminar Report

Impact of Isolation Levels and Connection Pooling on PostgreSQL Scalability under OLTP Load

Paul Marx

MatrNr: 14773463

Supervisor: Mojtaba Akbari

Georg-August-Universität Göttingen
Institute of Computer Science

September 29, 2026

Abstract

Database systems are widely used for online transaction processing, for example in web-shop applications. This can involve many short write transactions that access the same database rows and must be processed by the database. To coordinate such concurrent access, concurrency control can prevent certain anomalies and inconsistent data states. At the same time, stricter isolation levels may introduce additional coordination overhead as well as transaction retries or aborts. Database performance can also be negatively affected by the overhead of repeatedly establishing new connections to the database. This overhead can be reduced through the use of a connection pooler. The proxy maintains persistent connections to the database that can be assigned to transactions and reused.

The objective of this work is to investigate the scalability of PostgreSQL under increasing concurrency and OLTP load. In particular, the study examines the influence of connection pooling through the use of PgBouncer and the influence of the Read Committed, Repeatable Read, and Serializable isolation levels. Database performance is evaluated based on P95 latency, average latency, transactions per second, average waiting sessions, as well as transaction retries and failed transactions.

The results show that, when a new connection is established for each transaction, PgBouncer achieves higher throughput than direct database connections across all examined isolation levels. In addition, for persistent direct connections under high concurrency, Repeatable Read and Serializable show lower TPS and higher P95 latency than Read Committed. This is associated with higher failure rates and retry rates for these isolation levels.

Declaration on the use of AI in the context of examinations

In this work I have used AI as follows:

- ☐ Not at all
- ☒ During brainstorming
- ☐ When creating the outline
- ☐ To write individual passages, altogether to the extent of 0% of the entire text
- ☒ For the development of software source texts
- ☒ For optimizing or restructuring software source texts
- ☐ For proofreading or optimizing
- ☒ For the translation of the self-written German report into English

I hereby declare that I have stated all uses completely.

Missing or incorrect information will be considered as an attempt to cheat.

Contents

List of Figures	iv
List of Abbreviations	v
1 Introduction	1
2 Background	2
2.1 Snapshots and Multiversion Concurrency Control	2
2.2 Transaction Isolation Levels in PostgreSQL	2
2.3 Connection Pooling with PgBouncer	3
3 Related Work	4
4 Methodology	5
4.1 Performance Metrics	5
4.2 Independent Variables and Test Matrix	5
4.3 Transaction Logic and Workload Design	5
4.4 Tools and Automation	6
5 Results	7
5.1 Impact of Isolation Levels	7
5.2 Impact of Connection Pooling	8
6 Discussion	10
6.1 Concurrency and Lock Contention	10
6.2 Isolation-Dependent Retry and Abort Behavior	10
6.3 Connection Management and Pooling	11
6.4 Limitations	11
7 Conclusion	12
References	13
A Transparency Report: Tools and Third-Party Contributions	A1

List of Figures

1	Basic connection pooling setup with PgBouncer.	3
2	Benchmark setup and connection paths.	6
3	TPS and P95 latency across isolation levels using Direct Persistent connections.	7
4	Retry rate and average waiting sessions across isolation levels using Direct Persistent connections.	8
5	TPS and P95 latency under Read Committed for Direct Reconnect and PgBouncer Reconnect connections.	9
6	TPS and P95 latency under Read Committed for Direct Persistent and PgBouncer Persistent connections.	9

List of Abbreviations

DBMS Database Management System

GWDG Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen

MVCC Multiversion Concurrency Control

OLTP Online Transaction Processing

SI Snapshot Isolation

SSI Serializable Snapshot Isolation

TCP Transmission Control Protocol

TPS Transactions Per Second

1 Introduction

An important use case for the industrial application of databases is online transaction processing (OLTP) [Fek]. The workload is characterized by transactions that update only a small number of records [Sto+07], which, in the case of a webshop, may, for example, modify inventory levels or customer data [Fek]. These writing transactions are typically interspersed with small- to medium-sized queries, such as those that may arise when checking the status of an order [Sto+14]. However, the access distribution of an OLTP application is rarely uniform across all database rows. Statistically, individual tuples are accessed considerably more frequently than others, which results in contention [Yu+14]. Especially in modern OLTP applications, which are characterized by high transaction volumes [Sto+14], this can become a bottleneck. This applies in particular when the number of concurrent users increases. Consequently, modern databases require effective strategies as well as the ability to maintain high performance under increasing load, which can also be referred to as database scalability [Kle17, p. 8]. One method for improving database performance is connection optimization through connection pooling. In this approach, clients connect to the database through a proxy-managed connection pool that maintains, manages, and reuses persistent connections to the database [KM17]. In addition to increasing throughput, ensuring data integrity is of central importance. Even with numerous concurrent transactions, OLTP systems must ensure that defined constraints and application-specific invariants are not violated. Without concurrency control, anomalies and inconsistent data states can occur [Fek]. To prevent anomalies, stricter isolation levels can be configured for transaction execution. However, stricter isolation levels may require additional effort to detect and prevent read-write and write-write conflicts that can occur between concurrent transactions [Kle17, pp. 224–252].

This work contributes a systematic and experimental comparison of PostgreSQL under the influence of connection pooling and different isolation levels. For this purpose, a common OLTP workload is examined for the different influencing factors. Specifically, the study investigates and evaluates the differences between the Read Committed, Repeatable Read, and Serializable isolation levels as the number of concurrent clients increases. The study uses PgBouncer as a proxy, PostgreSQL as the DBMS, and a simple webshop on which the benchmark is performed. Database performance is measured primarily based on database latency and transactions per second (TPS). In addition, transaction retries and waiting sessions are recorded.

This leads to the following research question: How do transaction isolation levels and connection pooling affect throughput and latency in PostgreSQL under high OLTP load?

The results of this work show that, under direct persistent connections and high concurrency, the Repeatable Read and Serializable isolation levels exhibit higher retry rates, higher P95 latency, and lower throughput than Read Committed. When a new connection is established for each transaction, PgBouncer substantially increases TPS compared with direct connections to PostgreSQL.

The report is structured as follows. First, the Background chapter introduces the fundamental concepts of PostgreSQL and connection pooling. Subsequently, the current state of scientific research is reviewed with regard to the research question. After the exact methodology of the benchmark has been explained, the results are presented in the Results chapter and subsequently discussed in the Discussion chapter. The report concludes with a presentation of the most important findings in the Conclusion chapter.

2 Background

The Background chapter introduces the essential concepts required to understand and academically contextualize the topic of this project report. In particular, it presents the characteristics of the examined isolation levels, the mechanisms of Snapshot Isolation (SI) and Serializable Snapshot Isolation (SSI), as well as the general principle of connection pooling.

2.1 Snapshots and Multiversion Concurrency Control

Multiple concurrent and simple read accesses to the same database row are generally possible without major problems. However, when read and write accesses occur simultaneously, and especially when multiple write operations compete for the same database rows, inconsistent data may be read or blocking may occur. PostgreSQL addresses and reduces this problem through the use of Multiversion Concurrency Control (MVCC). MVCC is technically implemented by allowing each SQL statement to see a snapshot of the data representing a previous point in time [Groa]. The snapshot contains the data that had been committed at that point in time and therefore provides a consistent view of the data [Rog23, p. 80]. Consequently, the snapshot does not automatically change when the underlying data is modified in the meantime [Groa]. Each transaction uses its own snapshots, which means that different transactions can use different snapshots that were created at different points in time [Rog23, p. 80]. As a result, transactions can generally perform updates while other transactions read the same database rows, because the reading transaction can use a different version of the database row for its snapshot. The lifetime of a snapshot depends on the isolation level [Groa].

2.2 Transaction Isolation Levels in PostgreSQL

A central requirement for database management systems such as PostgreSQL is to ensure data consistency and data correctness [Rog23, p. 39]. The use of stricter isolation levels can help ensure that complex integrity constraints are preserved when transactions are executed concurrently [Rog23, pp. 39–40]. PostgreSQL supports the isolation levels Read Committed, Repeatable Read, and Serializable.

The isolation levels Read Committed and Read Uncommitted do not differ in their implementation in PostgreSQL and therefore behave identically [Grob]. At the lowest effectively implemented isolation level, Read Committed, reading uncommitted data is prevented, which avoids an anomaly referred to as a dirty read [Grob]. Read Committed uses snapshots that are taken at the beginning of a query and remain valid only for the duration of that query [GP12]. As a consequence, a transaction that executes the same query multiple times may read different values from the same database rows, referred to as a nonrepeatable read, or may observe a changed number of database rows, referred to as a phantom read, if updates by concurrent transactions have occurred in the meantime [Grob].

These anomalies can be prevented by the stricter Repeatable Read isolation level. PostgreSQL implements the Repeatable Read isolation level based on Snapshot Isolation [GP12]. Each transaction uses a consistent snapshot for the entire duration of the transaction, which is created before the first statement of the transaction is executed [GP12]. In the case of concurrent update attempts, the respective row is locked, while in other cases

queries can be executed concurrently without blocking each other [Rog23, p. 44]. In the case of parallel and conflicting updates, one of the involved transactions may additionally be rolled back with a serialization failure [Grob]. However, the serialization failure does not indicate that a serialization anomaly has occurred in this case. PostgreSQL aborts the transaction because it attempts to modify a row that has already been changed after the transaction’s own snapshot was created.

The strictest isolation level, Serializable, uses Serializable Snapshot Isolation to guarantee serializability and prevent serialization anomalies. In addition to Snapshot Isolation, further checks are performed for this purpose [GP12]. Serializable therefore detects conflict patterns at runtime that could lead to a nonserializable execution of transactions under SI. If such a conflict is detected, one of the involved transactions is aborted before a nonserializable execution can occur [CRF09]. In this way, simple write-skew anomalies can be effectively prevented [GP12].

2.3 Connection Pooling with PgBouncer

Connection pooling manages a set of reusable database connections that can be assigned to different clients [Tiw+16]. Through the use of connection pooling, database performance can be improved and user waiting time can be reduced [KM17]. By using resources more efficiently, the overhead caused by repeatedly establishing new connections to the database can be avoided.

Connection pools can be created when the server is started. As soon as a connection is required, the connection pool is searched for an available connection. If a connection is available, it is assigned to the requester [KM17]. Figure 1 illustrates the general connection setup in which multiple clients connect to PgBouncer and subsequently use a connection from the connection pool to access the database. If no connection is available, the requester is placed in a waiting queue. Depending on the configuration and implementation of the connection pool, an additional connection may also be established for the requester. After the connection has been used, it is not necessarily closed but is instead returned to the connection pool and made available for subsequent transactions [KM17]. Typically, the connection pooler can be configured using a range of parameters.

For the PostgreSQL connection pooler PgBouncer, particularly with regard to the workload examined in this study, the size of the connection pool and the pooling mode are especially important. The pooling mode determines how long PgBouncer assigns a PostgreSQL backend connection to a client. In Transaction Pooling mode, the PostgreSQL backend connection is assigned for the duration of a transaction. In addition to Transaction Pooling, PgBouncer can also be configured to use Session Pooling or Statement Pooling [Aut26].

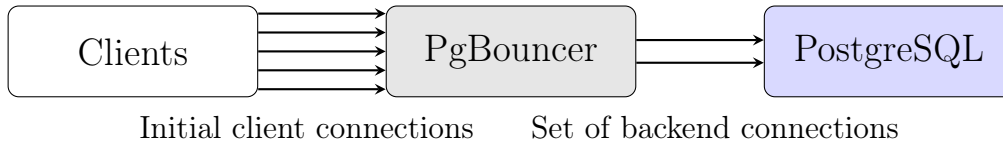


Figure 1: Basic connection pooling setup with PgBouncer.

3 Related Work

Mustafa Utku Kalay [Kal25] investigated concurrency challenges in database systems in 2025, with a focus on PostgreSQL. In addition to analyzing the current state of PostgreSQL, the study validated theoretical concepts, for example regarding Snapshot Isolation and Serializable Snapshot Isolation, through practical experiments. In the study, Repeatable Read corresponds to Snapshot Isolation, whereas Serializable is implemented using SSI. Different workloads, including OLTP workloads, are examined, and metrics such as TPS and resource utilization based on memory consumption are measured. As in our experimental setup, pgbench was used as the benchmarking tool.

A key finding of the paper is that Read Committed provides better performance for write-intensive OLTP workloads than Serializable and Repeatable Read. The main reasons for this difference are higher rollback rates and the additional overhead required to handle write-write and read-write dependencies between concurrent transactions. For larger data volumes, however, the paper identifies memory contention as a decisive factor influencing database TPS. Consequently, the relative importance of the performance overhead caused by SSI decreases for larger datasets. The performance differences between SSI and SI under different levels of concurrency were investigated in more detail by Fairouz Zendaoui and Walid Khaled Hidouci [ZH15]. Their experiments show that a higher number of concurrent transactions increases database throughput, while at the same time creating more dependencies between read-only and update transactions. In particular, for write-intensive workloads, SI and SSI are found to cause transaction aborts more frequently, which increases the abort rate and reduces throughput. Under high read-write contention, the throughput of SSI was substantially lower than that of SI. Here as well, the additional overhead required for conflict detection and for ensuring serializability is identified as the cause.

The scalability of databases through connection pooling was investigated by Butrovich et al. in 2023 [But+23]. In their study, the performance of PostgreSQL was benchmarked using several different OLTP workloads. For comparison, the authors examined direct connections to the database, Tigger, a proxy based on PgBouncer, as well as several open-source PostgreSQL-Scompatible proxies such as PgBouncer and Odyssey. The results show that P50 and P99 latency for serverless clients, which establish a new connection for each transaction, are substantially reduced when a proxy is used instead of a direct database connection. In addition, a no-operation workload was tested in which the proxies were executed on the same server as the DBMS in order to avoid network hops. In transaction pooling mode over TCP, PgBouncer achieved 71 % lower TPS than a direct connection to the database for this specific setup. This demonstrates that connection pooling itself can also introduce additional overhead. The authors attribute the proxy overhead to PgBouncer’s single-threaded architecture, system calls, and data copies between user space and kernel space. In another high-concurrency workload, Jayraj Kiran Ladkat and Pranati Waghodekar [LW25] observed a positive influence of connection pooling on PostgreSQL scalability. In their benchmark, latency was reduced by 30 % compared with direct connections, while TPS increased by 44 %.

The studies presented above predominantly focus on connection pooling and isolation levels separately. In contrast, our benchmark specifically investigates the influence of the isolation levels and connection pooling across different numbers of concurrent clients using a common OLTP workload.

4 Methodology

At the center of this work is the research question of how isolation levels and connection pooling affect both throughput and latency in PostgreSQL under OLTP load.

4.1 Performance Metrics

Both metrics were collected using `pgbench`, a program specifically designed for conducting benchmarks on PostgreSQL [Groe]. Average Latency and throughput were parsed directly from the `pgbench` benchmark output, while P95 latency was calculated separately based on the per-transaction logs. These logs were generated using the `log` option of `pgbench` and contain additional information about each individual transaction executed during the benchmark [Groe]. In addition, failed and retried transactions were collected from the `pgbench` benchmark output for every benchmark run. Average waiting sessions were also determined as an additional metric. For this purpose, the `pg_stat_activity` view was queried every 2 seconds during the execution of the benchmark. The view contains one row per server process and provides information about the current activity of the corresponding process. The number of sessions currently waiting for a lock was determined by filtering the `wait_event_type` column [Grod]. Subsequently, the average number of waiting sessions over the complete duration of each benchmark run was calculated.

4.2 Independent Variables and Test Matrix

The performance metrics were examined for different numbers of concurrent clients. Benchmark runs were conducted with 1, 10, 25, 50, 100, and 200 clients in order to expose PostgreSQL to OLTP loads with different levels of intensity.

In addition, a specific isolation level was configured for every benchmark run. The isolation levels investigated in the benchmark were Read Committed, Repeatable Read, and Serializable.

With regard to the connection to the database, a general distinction was made between direct connections to PostgreSQL and connections managed through PgBouncer. The connections were further divided into persistent connections and connections that were newly established for each individual transaction. Consequently, the four connection modes Direct Persistent, Direct Reconnect, PgBouncer Persistent, and PgBouncer Reconnect were distinguished throughout the benchmark.

Considering the six different client counts, the three investigated isolation levels, and the four different connection modes, the complete test matrix consisted of 72 different benchmark configurations. Each individual benchmark run had a duration of 90 seconds. Three independent measurements were performed for every configuration, resulting in a total of 216 benchmark runs. For each performance metric and for each benchmark configuration, the arithmetic mean of the three independent measurements was calculated.

4.3 Transaction Logic and Workload Design

To execute the workloads, a simple webshop database was set up with the tables Products, Stock, Orders, Customers, and a mapping table between Orders and Products. This database structure made it possible to simulate fundamental activities of a webshop at the database level. In total, six different SQL scripts were used for the workload. These

included, for example, purchasing a product, updating inventory levels, querying the price of a product, and performing price adjustments. The workload also deliberately contains transactions that, under high concurrency, can produce anomalies such as a nonrepeatable read or a serialization anomaly. This design allows the workload to differentiate between the isolation levels based on their different behavior under concurrent execution. To prevent highly conflicting activities from completely dominating the workload under high contention, the individual webshop activities are selected according to adjusted probabilities.

4.4 Tools and Automation

The entire execution process of the benchmark is orchestrated by a Python script. The script defines the individual configurations for the different benchmark runs and iterates over all benchmark runs in the test matrix. In addition, the script invokes the `pgbench` benchmarking program and parses the required values in order to subsequently store the performance metrics or calculate additional metrics where necessary. `Pgbench` executes the benchmark run and, as illustrated in Figure 2, connects either directly to PostgreSQL or first to PgBouncer. The maximum number of execution attempts per transaction was set to 5. The PgBouncer connection pool was limited to 25 backend connections. Consequently, PgBouncer was able to serve a larger number of concurrent clients through this limited connection pool. The webshop database itself was hosted on PostgreSQL running on its own virtual machine. The benchmark script and `pgbench` were executed on a separate virtual machine in order to minimize the influence of benchmark orchestration and metric calculation on the measured database performance.

Ubuntu 22.04.5 LTS was used on both virtual machines. The version used for both `pgbench` and PostgreSQL was 16.14. PgBouncer was used in version 1.25.2.

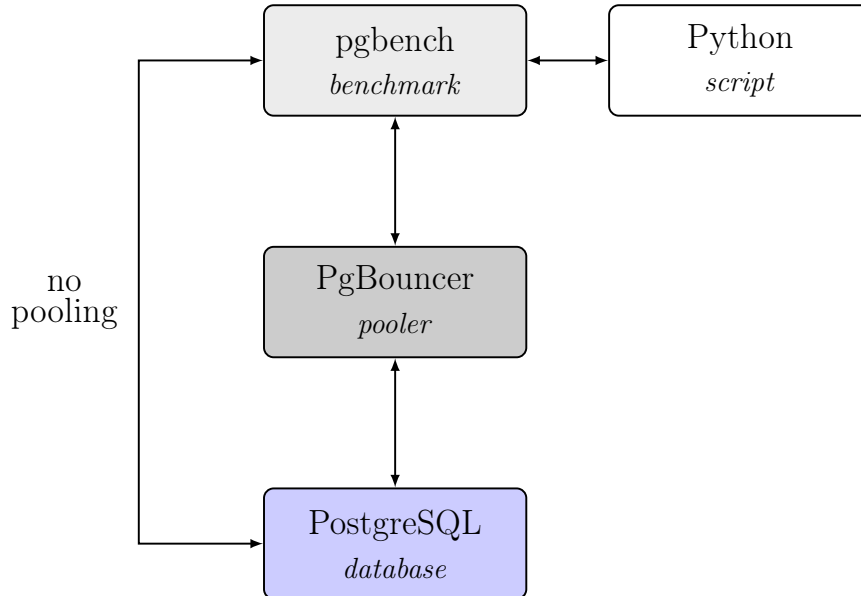


Figure 2: Benchmark setup and connection paths.

5 Results

To evaluate the performance of PostgreSQL under OLTP load, the metrics transactions per second, P95 latency, average waiting sessions, as well as transaction failure and retry rates were examined in particular. For each metric, the arithmetic mean across three test runs was calculated. The benchmark results are presented primarily with regard to the influence of the isolation levels and the type of database connection.

5.1 Impact of Isolation Levels

Figure 3 shows both TPS and P95 latency as a function of the number of concurrent clients. The metrics refer to direct and persistent connections to the database. For all three isolation levels, TPS increases as the number of clients rises up to 100 clients. With a TPS of 2958.7 at 100 clients, Read Committed achieves higher values than Repeatable Read ($\Delta +204.6$ TPS) and Serializable ($\Delta +958.4$ TPS). When the number of clients is increased to 200, throughput decreases to 2177.2 TPS for Repeatable Read and to 1510.5 TPS for Serializable. In contrast, the throughput of Read Committed continues to increase slightly. At 200 clients, Read Committed reaches the highest measured TPS of 3239.4.

P95 latency increases for all isolation levels as the number of concurrent clients rises. At 200 clients, Read Committed exhibits not only the highest throughput but also the lowest P95 latency at 107.9 ms. Serializable shows the highest P95 latency at 200 clients, reaching 268.8 ms. At 200 clients, Repeatable Read achieves a lower P95 latency than Serializable at 219.4 ms ($\Delta -49.4$ ms).

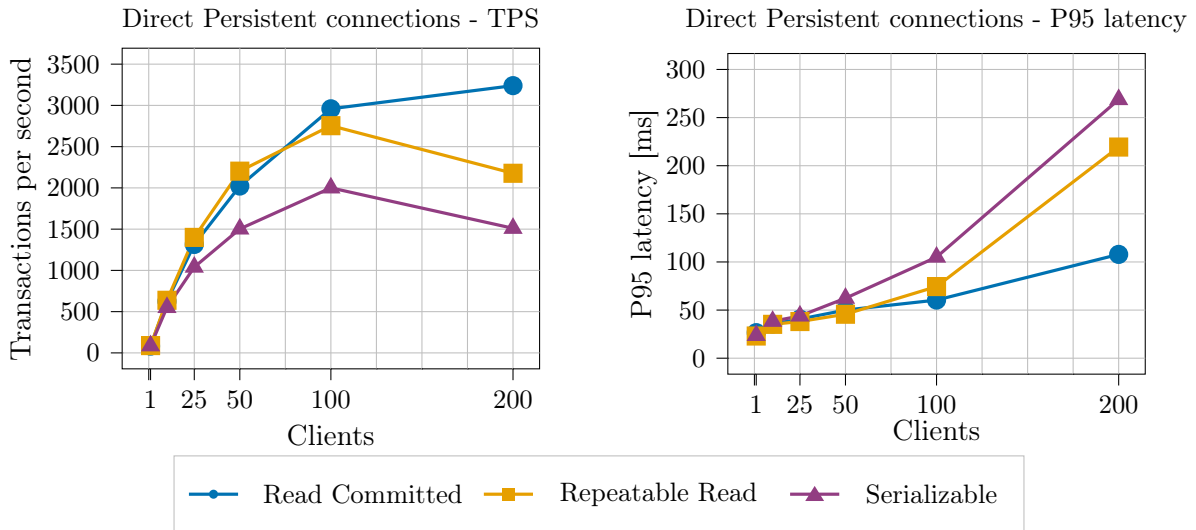


Figure 3: TPS and P95 latency across isolation levels using Direct Persistent connections.

With regard to the retry rate, Figure 4 shows clear differences between the investigated isolation levels. Under Read Committed, no transactions are retried or fail for the given workload. For Repeatable Read, the retry rate increases steadily as the number of concurrent clients rises and reaches its maximum of 21.8 % at 200 clients. Under the Serializable isolation level, the highest retry rate is observed across all client counts. At only 25 clients, the retry rate already increases to 39.2 % and remains at a consistently high

level between 34.0 % and 41.6 % as the number of clients increases further. At 200 clients, 7.9 % of all transactions under Serializable ultimately fail after the maximum number of four retries has been reached.

With regard to average waiting sessions, the isolation levels show almost identical behavior up to a client count of 100. On average, no more than 2.3 sessions are waiting for lock events within this range. At 200 clients, the hot row contention of the workload increases substantially. At the same time, an increase in average waiting sessions can be observed for all isolation levels. Under Read Committed, the highest number of waiting sessions is observed at 200 clients, with an average value of 29.4. With this value, Read Committed is approximately 9.3 waiting sessions above Repeatable Read and 6.1 waiting sessions above Serializable.

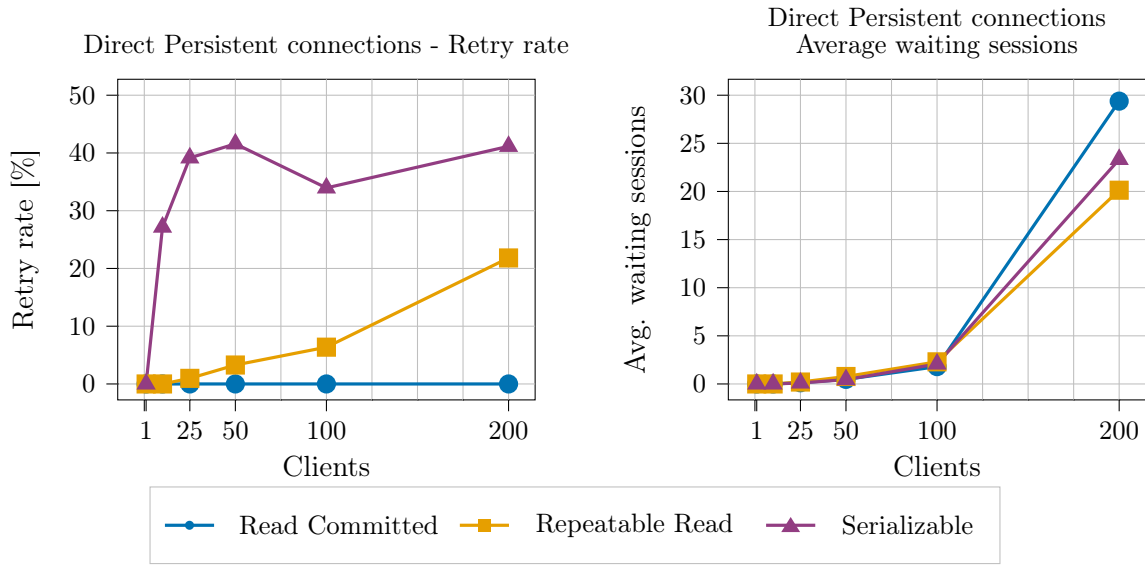


Figure 4: Retry rate and average waiting sessions across isolation levels using Direct Persistent connections.

5.2 Impact of Connection Pooling

Figure 5 shows TPS and P95 latency for connection modes in which a new client connection is established for every transaction. A distinction is made between direct connections to the database (Direct Reconnect) and connections established through PgBouncer (PgBouncer Reconnect). The results were evaluated for the Read Committed isolation level. At 25 concurrent clients, PgBouncer already reaches a TPS of 793.7 and therefore achieves approximately 7.1 times the throughput of direct connections to the database. As the number of concurrent clients increases further, TPS for PgBouncer remains within a range of 814.3 to 768.5 and consistently stays above the throughput achieved with direct connection establishment. At 200 clients, direct connections reach a TPS of 108.9, which corresponds to approximately 660 fewer transactions per second than PgBouncer.

P95 latency increases for both connection modes as the number of concurrent clients rises. Starting at 10 concurrent clients, connections through PgBouncer show a substantially lower P95 latency than direct connections to the database. At 200 concurrent clients, P95 latency for connections through PgBouncer is 353.4 ms. At this load level, this value is 1867.5 ms lower than the P95 latency measured for a direct database connection.

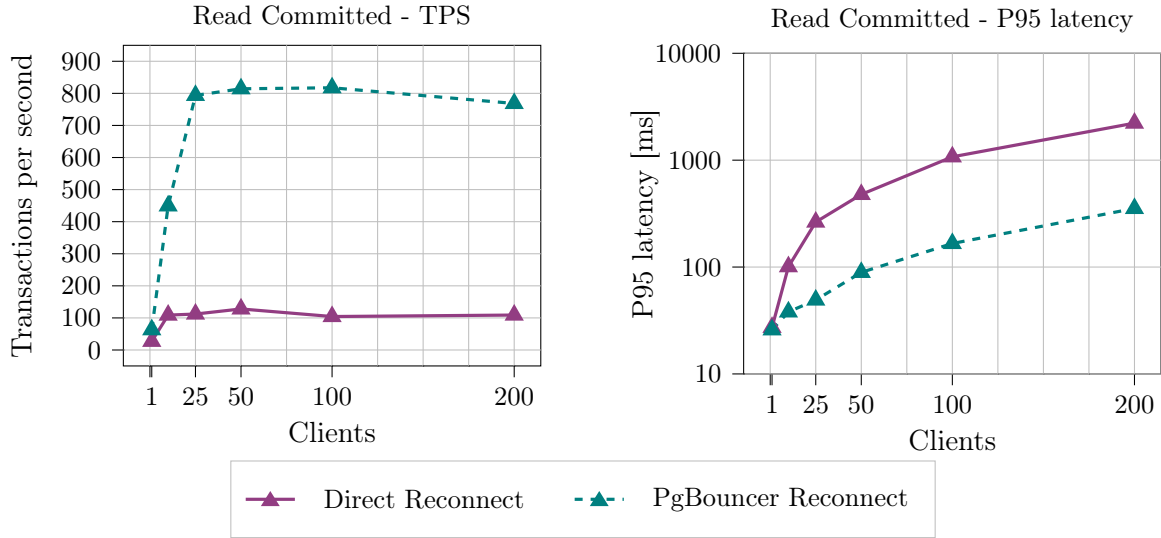


Figure 5: TPS and P95 latency under Read Committed for Direct Reconnect and PgBouncer Reconnect connections.

When persistent client connections are used, only minor differences between PgBouncer and direct database connections can be observed for up to 25 concurrent clients, as shown in Figure 6. At 25 clients, direct connections achieve 1314.5 TPS, while PgBouncer reaches 1347.8 TPS. As the number of concurrent clients increases further, the throughput achieved with PgBouncer reaches a plateau, whereas persistent direct connections continue to increase and reach a throughput of 3239.4 TPS at 200 clients ($\Delta +2002.5$ TPS). At the same time, P95 latency for PgBouncer reaches 248.3 ms at 200 clients and is therefore 140.5 ms higher than for persistent direct client connections.

Compared with newly established client connections, persistent connections achieve both higher throughput and lower P95 latency at higher numbers of concurrent clients.

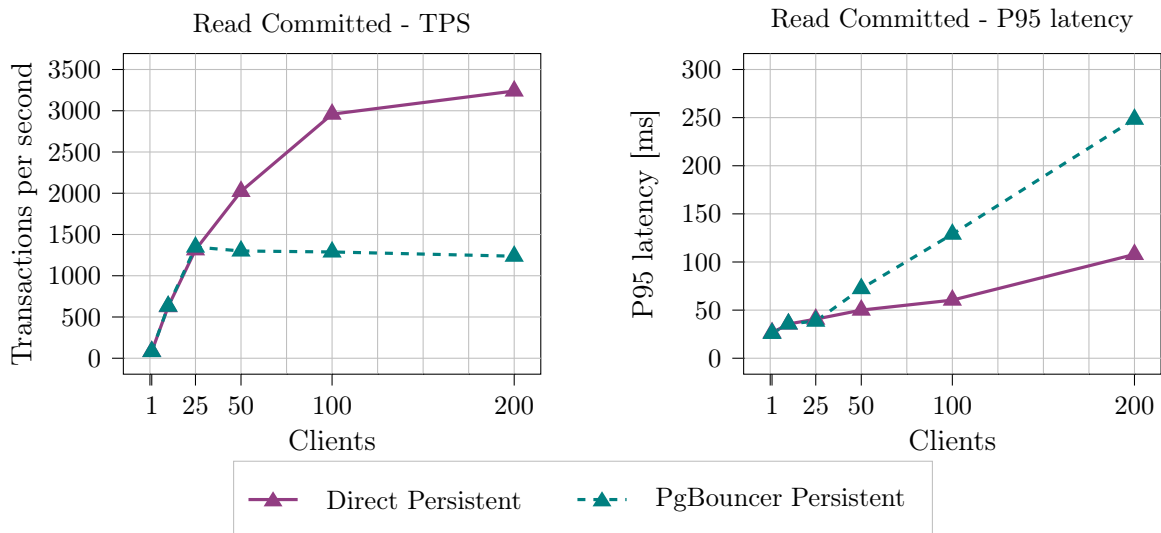


Figure 6: TPS and P95 latency under Read Committed for Direct Persistent and PgBouncer Persistent connections.

6 Discussion

6.1 Concurrency and Lock Contention

A key component of this work is the investigation of database performance with an increasing number of concurrent clients. For the examined isolation levels Read Committed, Repeatable Read, and Serializable, throughput increased substantially with persistent direct connections up to 100 clients. The higher number of concurrent clients enables more transactions to be processed within a shorter period of time. As long as other resources, such as CPU capacity, storage operations, or row contention, do not significantly limit performance, a larger number of clients also results in a higher number of successfully executed transactions per second. Under the Read Committed isolation level, by contrast, TPS increases only slightly by 9.4 % when the number of concurrent clients is doubled from 100 to 200 with direct persistent connections. Across all examined client counts, however, no transactions under Read Committed fail permanently or require retries.

The fact that TPS reaches a plateau is consistent with an increasing average number of sessions waiting for locks. In general, PostgreSQL’s implementation of Multiversion Concurrency Control ensures that read and write accesses to the same database row do not block each other [Groa]. During a row update, the database row is instead locked against other writing transactions until the transaction has completed [Groc]. In the workload used for the webshop scenario, row updates at 100 clients were predominantly distributed across different products. At 200 clients the same products and inventory records were updated more frequently, increasing contention for the same rows. Consequently, the probability of mutually blocking updates also increased, causing the average number of waiting sessions to rise substantially. Because transactions had to wait until the database rows were released, P95 latency increased substantially, while TPS stagnated.

For Repeatable Read and Serializable, lower TPS can be observed at 200 clients with direct persistent connections. At the same time, the average number of waiting sessions is also lower for both isolation levels than for Read Committed. Repeatable Read and Serializable additionally incur overhead from failed and retried transactions.

6.2 Isolation-Dependent Retry and Abort Behavior

The transaction abort rate differs fundamentally across the three isolation levels for the examined OLTP workload and for direct persistent connections. As already shown, no transaction retries occurred under the Read Committed isolation level. Under Repeatable Read, the number of transaction retries increases with an increasing number of concurrent clients. This is caused by the snapshot isolation under which Repeatable Read transactions are processed [Grob]. As the number of concurrent clients increases, the probability also increases that a concurrent transaction has already updated the same row in the meantime and successfully committed. Consequently, the number of transactions that abort with a serialization failure increases. Under Serializable, SSI additionally detects conflict patterns that may cause a serialization anomaly and aborts involved transactions [Grob]. Under the Serializable isolation level, both the retry rate and the rate of transactions that ultimately fail are therefore the highest. The observed result of our OLTP workload is consistent with the findings of Zendaoui and Hidouci [ZH15], who likewise observed higher abort rates under SSI with an increasing number of conflicts and, consequently, a reduction in TPS.

It can be concluded that, with a high number of concurrent clients, throughput decreases for Repeatable Read and Serializable, while the latency of successfully committed transactions increases. At the same time, however, the stricter isolation levels provide stronger consistency guarantees for concurrent transactions and, in the case of Serializable, even guarantee serializability.

6.3 Connection Management and Pooling

When the connection to the database or to PgBouncer had to be established again, connection pooling resulted in a substantial increase in throughput. As soon as multiple concurrent clients execute transactions simultaneously, PgBouncer can reduce connection establishment overhead by reusing existing backend connections to the database. Reusing connections from the connection pool also reduces P95 latency for all higher client counts. Thus, our experiment generally confirms the performance improvement provided by connection pooling in terms of TPS and latency, as also reported by Ladkat and Waghodekar [LW25].

When considering persistent connections, another characteristic of connection pooling becomes apparent. With persistent connections, TPS with PgBouncer stagnates from 25 clients onward compared with direct connections to the database. This can be explained by the selected connection pool size, which is limited to 25 backend connections to the database in our setup. Consequently, with more than 25 concurrent clients, the limited number of available backend connections provided by PgBouncer itself becomes a bottleneck. This is also reflected in the increasing P95 latency from 25 clients onward.

This demonstrates that connection pooling can reduce connection establishment overhead on the one hand, while at the same time limiting the possible concurrency through the maximum number of connections available in the connection pool.

6.4 Limitations

In addition to the examined influencing factors, no separate CPU, storage, or network measurements were performed. Likewise, our workload did not measure the overhead introduced by the use of PgBouncer itself, as investigated in isolation, for example, by Butrovich et al. [But+23]. In addition, the average number of waiting sessions was queried from the database every 2 seconds during the benchmark. This introduces a certain degree of inaccuracy because lock events of waiting sessions occurring between two measurement points are not necessarily captured.

The presented results revealed trends regarding the isolation levels and the use of connection pooling. However, it is important to note that the results are specific to the selected configuration and the OLTP workload used. Each parameter and design decision, such as the maximum number of retries, the number of threads, the PgBouncer configuration, or the selection of hot rows in the database, can influence the results.

Another limitation is that the presented benchmark on the webshop database examined only the technical performance and scalability of the database. Compared with a real webshop, not only successful or failed transactions would be relevant, but especially the correctness of the respective business processes. For example, the fact that the Repeatable Read isolation level prevents a nonrepeatable read is not directly reflected in throughput and latency. The trade-off between performance and data consistency would therefore have to be evaluated in a real webshop.

7 Conclusion

The report investigated the influence of the Read Committed, Repeatable Read, and Serializable isolation levels as well as the use of connection pooling on the scalability of PostgreSQL under OLTP load. Performance metrics such as P95 latency, average latency, and TPS were measured for different numbers of concurrent clients. This made it possible to evaluate the behavior of the isolation levels and connection pooling under increasing contention and concurrency using a common workload.

With regard to the isolation levels, performance with direct persistent connections deteriorated under high concurrency for Repeatable Read and Serializable compared with Read Committed. At 200 concurrent clients, Read Committed achieved a TPS of 3239.4, while Repeatable Read achieved a TPS that was 1062.2 lower. For the same configuration, Serializable showed a TPS that was another 666.7 lower than Repeatable Read. The lower throughput of the two stricter isolation levels was associated with a higher number of aborted and retried transactions. With direct persistent connections and 200 clients, 7.9 % of all transactions under Serializable ultimately failed. As the number of concurrent clients and the level of concurrency increased, particularly due to write transactions accessing the same database rows, a substantial increase in average waiting sessions and P95 latency was observed for persistent connections.

When connections were newly established for each transaction, the examined OLTP workload showed a substantial improvement in database performance when PgBouncer was used. At 25 clients, connection pooling increased TPS by a factor of 7.1 compared with direct connections, while P95 latency remained lower at the same time. The performance advantage of PgBouncer can be explained by the reuse of backend connections, which reduces the overhead of repeatedly establishing new connections. When the workload was instead executed using persistent connections to PgBouncer or directly to the database, the results showed that PgBouncer itself could become a bottleneck. In this configuration, the connection pool, which was configured with 25 backend connections, no longer provided a performance advantage over persistent direct connections when more than 25 clients were running concurrently.

Overall, the study showed that both PgBouncer and the choice of isolation level can have a substantial influence on the scalability of PostgreSQL under OLTP load. However, the observed trends apply primarily to the specific workload and tool configuration used, since changes such as adjusting the connection pool size can also affect the results.

Future work could extend the existing benchmark setup by measuring additional important influencing factors. For example, the impact of and bottlenecks caused by I/O, network, or CPU resources could be investigated specifically.

References

- [Aut26] PgBouncer Authors. *PgBouncer Documentation, Chapter Usage*. 2026. URL: <https://www.pgouncer.org/usage.html> (visited on 09/18/2026).
- [But+23] Matthew Butrovich et al. “Tigger: A Database Proxy That Bounces with User-Bypass”. In: *Proceedings of the VLDB Endowment* 16.11 (July 2023), pp. 3335–3348. ISSN: 2150-8097. DOI: 10.14778/3611479.3611530. URL: <https://dl.acm.org/doi/10.14778/3611479.3611530> (visited on 09/18/2026).
- [CRF09] Michael J. Cahill, Uwe Roehm, and Alan D. Fekete. “Serializable Isolation for Snapshot Databases”. In: *ACM Transactions on Database Systems* 34.4 (Dec. 2009), pp. 1–42. URL: <https://dl.acm.org/doi/abs/10.1145/1620585.1620587> (visited on 09/19/2026).
- [Fek] Alan Fekete. “Allocating Isolation Levels to Transactions”. In: SIGMOD/PODS05: International Conference on Management of Data and Symposium on Principles of Database Systems (Baltimore, Maryland, USA). 06/2005, pp. 206–215. URL: <https://dl.acm.org/doi/10.1145/1065167.1065193> (visited on 09/18/2026).
- [GP12] Kevin Grittner and Dan R. K. Ports. “Serializable Snapshot Isolation in PostgreSQL”. In: *Proceedings of the VLDB Endowment* 5.12 (Aug. 2012), pp. 1850–1861. URL: <https://arxiv.org/abs/1208.4179> (visited on 09/18/2026).
- [Groa] PostgreSQL Global Development Group. *PostgreSQL Documentation, Chapter 13.1. Introduction*. URL: <https://www.postgresql.org/docs/16/mvcc-intro.html> (visited on 09/20/2026).
- [Grob] PostgreSQL Global Development Group. *PostgreSQL Documentation, Chapter 13.2. Transaction Isolation*. URL: <https://www.postgresql.org/docs/16/transaction-iso.html> (visited on 09/20/2026).
- [Groc] PostgreSQL Global Development Group. *PostgreSQL Documentation, Chapter 13.3.2. Row-Level Locks*. URL: <https://www.postgresql.org/docs/16/explicit-locking.html> (visited on 09/20/2026).
- [Grod] PostgreSQL Global Development Group. *PostgreSQL Documentation, Chapter 27.2.3. Pg_stat_activity*. URL: <https://www.postgresql.org/docs/current/monitoring-stats.html#MONITORING-PG-STAT-ACTIVITY-VIEW> (visited on 09/18/2026).
- [Groe] PostgreSQL Global Development Group. *PostgreSQL Documentation, Chapter Client Applications*. URL: <https://www.postgresql.org/docs/16/pgbench.html> (visited on 09/23/2026).
- [Kal25] Mustafa Utku Kalay. “Concurrency Challenges in Database Systems: A Focus on PostgreSQL”. In: *Journal of Computer Science and Technologies* 6.1 (July 2025), pp. 1–16. URL: <https://dergipark.org.tr/en/pub/bibtet/article/1574178> (visited on 09/19/2026).
- [Kle17] Martin Kleppmann. *Designing Data-Intensive Applications*. O’Reilly Media, Inc., Apr. 2017.

- [KM17] Gupta Kirti and Mathuria Manish. “Improving Performance of Web Application Approaches Using Connection Pooling”. In: 2017 International Conference of Electronics, Communication and Aerospace Technology (ICECA). Coimbatore, India: IEEE, Apr. 2017. URL: <https://ieeexplore.ieee.org/abstract/document/8212833> (visited on 09/18/2026).
- [LW25] Jayraj Kiran Laddkat and Pranati Waghodekar. “Real-World Scalability of PostgreSQL: Practical Techniques Use Case Study”. In: *2025 IEEE International Conference on Blockchain and Distributed Systems Security (ICBDS)*. Kolhapur, India: IEEE, Oct. 2025. URL: <https://ieeexplore.ieee.org/abstract/document/11377486> (visited on 09/18/2026).
- [Rog23] Egor Rogov. *PostgreSQL 14 Internals*. Moscow: Postgres Professional, 2023. URL: https://edu.postgrespro.com/postgresql_internals-14_en.pdf (visited on 09/20/2026).
- [Sto+07] Michael Stonebraker et al. “The End of an Architectural Era (It’s Time for a Complete Rewrite)”. In: Proceedings of the 33rd International Conference on Very Large Data Bases. Vienna, Sept. 2007, pp. 1150–1160. URL: <https://dl.acm.org/doi/10.1145/3226595.3226637> (visited on 08/26/2026).
- [Sto+14] Michael Stonebraker et al. “Enterprise Database Applications and the Cloud: A Difficult Road Ahead”. In: IEEE International Conference on Cloud Engineering (IC2E). Boston, MA, USA: IEEE, Mar. 2014. URL: <https://ieeexplore.ieee.org/abstract/document/6903452> (visited on 09/18/2026).
- [Tiw+16] Ankit Kumar Tiwari et al. “Performance Optimization of Web Applications Using Connection Pooling”. In: International Conference on Innovations in Information, Embedded and Communication Systems. 2016. URL: https://www.academia.edu/25482164/Performance_Optimization_of_Web_Applications_using_Connection_Pooling (visited on 09/23/2026).
- [Yu+14] Xiangyao Yu et al. “Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores”. In: *Proceedings of the VLDB Endowment* 8.3 (Nov. 2014), pp. 209–220. URL: <https://dl.acm.org/doi/10.14778/2735508.2735511> (visited on 09/18/2026).
- [ZH15] Fairouz Zendaoui and Walid Khaled Hidouci. “Performance Evaluation of Serializable Snapshot Isolation in PostgreSQL”. In: *2015 12th International Symposium on Programming and Systems (ISPS)*. Algiers, Algeria: IEEE, Apr. 2015. URL: <https://ieeexplore.ieee.org/abstract/document/7244971> (visited on 09/19/2026).

A Transparency Report: Tools and Third-Party Contributions

- Writing
 - LaTeX was used to create the report.
 - The computer science thesis template adapted by Jonathan Decker was used as the template.
 - TikZ was used to create all figures.
- Code
 - SQL scripts were used for the benchmark workload.
 - Bash scripts were used for the configuration and setup of the PostgreSQL database.
 - A Python script was used to orchestrate the benchmark and calculate the performance metrics.
 - Git was used for version control and to provide the source code and complete benchmark results.
- Database and Benchmark Tools
 - pgbench 16.14 was used to generate the benchmark workload.
 - PostgreSQL 16.14 was used as the DBMS.
 - PgBouncer 1.25.2 was used for connection pooling.
- AI
 - ChatGPT GPT-5.6 Sol was used for the purposes specified in the AI usage declaration.
- Third-Party
 - Mojtaba Akbari, as the supervisor of the project seminar, provided feedback and assistance in the preparation of the report and the associated presentation.
 - GWDG provided the virtual machines and technical infrastructure used to conduct the benchmark experiments.