BİLGİSAYAR BİLİMLERİ VE TEKNOLOJİLERİ DERGİSİ
JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGIES
http://dergipark.org.tr/tr/pub/bibted

Derleme Makale

Concurrency Challenges in Database Systems: A Focus on Postgresql

Mustafa Utku Kalay[*1]

[1] *Yıldız Technical University, Faculty of Electrical and Electronics, Computer Engineering Department, 34220, İstanbul, Turkiye*

Anahtar Kelimeler:
Concurrency
Isolation Levels
System Performance

ABSTRACT

In database systems, the concurrent execution of transactions has significant benefits for system performance. However, since concurrency can lead to various anomalies, it is crucial to correctly identify the concurrency settings that balance system efficiency and data integrity. Isolation levels are database-side settings that manage anomalies that threaten integrity. This article provides an overview of concurrency control techniques that ensure full accuracy, specifically serializability. Additionally, it discusses the importance of weaker isolation levels, particularly Snapshot Isolation (SI), which is preferred and increasingly adopted in many systems, while also addressing how it compromises accuracy. Methods that are SI-based but aim for full serializability are also examined. This review aims to combine theory and practice, focusing on the key principles of Serializable Snapshot Isolation (SSI) implemented under Multi-Version Concurrency Control (MVCC) in PostgreSQL. The promises of accuracy and performance mentioned are interpreted through some experimental findings. We believe that our article may serve as a motivation for researchers interested in this field and as an important resource for database administrators in the industry.

Veritabanı Sistemlerinde Eşzamanlılık Problemine Çözüm Önerileri Ve Postgresql Odaklı Ölçme Ve Değerlendirmeler

Keywords:
Eşzamanlılık
Yalıtım Seviyeleri
Sistem Performansı

ÖZ

Veritabanı sistemlerinde hareketlerin eşzamanlı çalışmasının sistem performansına önemli bir faydası vardır. Ancak, eşzamanlılık bazı anomalilere yol açtığı için, eşzamanlılık ayarının sistem verimi ve veri bütünlüğü arasındaki dengeyi sağlayarak doğru bir şekilde tespit edilmesi oldukça önemlidir. Yalıtım seviyeleri, özellikle bütünlüğü tehdit eden anomalileri yöneten veritabanı tarafı ayarlarıdır. Bu makalede, tam doğruluğu, yani serilenebilirliği sağlayan eşzamanlılık kontrol tekniklerine genel bir bakış verilmektedir. Bununla beraber, performansı artırmak için kullanılan daha zayıf yalıtım seviyelerinden, özellikle birçok sistemde tercih edilen ve zamanla yaygınlaşan Snapshot Isolation (SI)'nın önemi, fakat doğruluktan nasıl ödün verdiği üzerinde durulmaktadır. SI esaslı fakat tam serilenebilir olma yolundaki yöntemler de incelenmektedir. Teori ve pratiği birleştirmeyi hedefleyen bu inceleme makalemizde, PostgreSQL'de Çok Sürüm Eşzamanlı Kontrol (MVCC) çerçevesinde gerçekleştirilen Serializable Snapshot Isolation (SSI) yönteminin ana esasları incelendikten sonra, bahsedilen doğruluk ve performans vaatleri bazı deneysel çıktılar ile yorumlanmaktadır. Makalemizin bu konuda çalışmak isteyen araştırmacılar için bir motivasyon ve sektördeki veritabanı yöneticileri için de önemli bir kaynak olabileceğini düşünüyoruz.

[*]**Sorumlu Yazar**

[*](ukalay@yildiz.edu.tr) ORCID ID 0000-0002-8002-0268

e-ISSN: 2717-8579

Geliş Tarihi: 26/10/2024; Kabul Tarihi: 25/04/2025

1. INTRODUCTION

Database systems process transactions both indivisibly (atomically) and concurrently. While indivisibility is essential for maintaining data integrity, concurrency plays a crucial role in optimizing system throughput. The challenge of balancing these constraints primarily stems from the physical storage medium, often HDD or SSD, acting as a potential bottleneck in the system. The database recovery module ensures data integrity by confirming that all disk writing actions within a transaction are either permanently stored or no changes are made. Meanwhile, the concurrency module facilitates more efficient utilization of system resources, enhancing overall system performance through the interleaving of various disk read/write actions across all transactions (Silberschatz et al, 2019).

Ensuring data integrity by preventing, detecting, and rectifying issues arising from simultaneous access to shared data by multiple transactions is crucial. Transferring this responsibility to the concurrency module within the database system simplifies the task for application programmers. While application-level programmers can generally execute serializable transactions to ensure complete accuracy, meeting performance expectations may pose a challenge. This performance gap primarily arises from two reasons. Firstly, the database system expends effort detecting conflicts like read-write and write-write among active transactions. Secondly, time is lost due to aborted transactions resulting from detected anomalies. A prompt solution involves allowing transactions to operate at weaker isolation levels, potentially boosting application performance. However, this compromise increases the risk of anomalies and data integrity violations due to concurrent reading and writing of shared data. Consequently, detecting and rectifying these violations at the application level becomes crucial, adding significant workload (Kleppmann, 2017).

In summary, accurately determining the most suitable isolation level, considering both performance and accuracy, is thus challenging. If an application programmer executes operational modules as strong-isolation-level transactions, they must anticipate aborted transactions. Alternatively, adopting weaker isolation levels necessitates detecting and resolving potential integrity issues at the application level.

Even in today's widely used database systems, achieving expected accuracy in all cases remains challenging due to the emergence of complex and new types of anomalies in concurrent execution environments. Debugging becomes particularly difficult as catching rare anomalies and reproducing them for debugging purposes is relatively arduous (Kingsbury & Alvaro, 2020) and (Li et al, 2024).

In our study, we examined concurrency issues from both conceptual and experimental perspectives. After discussing the fundamental principles of concurrent execution and the problems it introduces, we shared some simple experimental results that validate the mentioned principles. Thus, we have conducted an analysis of the current state of PostgreSQL, a widely used database system, in this regard. Our measurements include system throughput (transactions per second, TPS), resource utilization (such as memory consumption), and accuracy across different workloads. We also explore edge-case scenarios designed to assess the impact of isolation levels on system performance. In our experimental findings based on centralized databases, in addition to validating known facts for SI and SSI, a significant result is that the performance overhead of SSI isolation diminishes as the dataset size increases.

In summary, this review-based study serves the following three purposes:

- A general examination and comparison of concurrency principles in modern databases.
- A better understanding of the increasingly popular Snapshot Isolation and Serializable SI approaches.
- Performance evaluation and anomaly detection in the concurrent execution of different types of workloads, such as OLTP and OLAP, in PgSQL.

Section 2 reviews related studies. Section 3 provides an overview of both traditional and contemporary approaches to isolation levels, specifically focusing on serializability, and addresses implementation issues in concurrency. In Section 4, we delve into the principles governing the implementation of the MVCC model and isolation levels in the PgSQL system. Additionally, we outline the core concepts and promises of Snapshot Isolation (SI) and Serializable Snapshot Isolation (SSI). Finally, in Section 5, we present anticipated results derived from measuring TPS rates through concurrent execution tests under varying workloads and isolation levels in PgSQL. We also provide insights based on experimental outputs, emphasizing system throughput and accuracy, particularly in the context of larger data sets. Section 6 concludes the paper.

2. RELATED STUDIES

In this section, we will examine the relevant literature in two groups. **The first group of studies** typically focus on comparing database systems, including NoSQL and relational databases, providing insights for selecting databases tailored to specific application scenarios. For instance, Almeida et al. (2023) evaluated the performance of four database systems—MySQL, PostgreSQL, Redis, and Memcached—using the YCSB benchmark under three workload types: update-heavy, read-only, and read-modify-writes. Their findings indicate that

NoSQL databases like Redis and Memcached excel at handling large datasets and offer superior scalability compared to relational databases such as MySQL and PostgreSQL. However, when scaling the number of threads in workloads, relational databases (MySQL and PostgreSQL) outperform NoSQL systems in throughput. This result confirms that relational databases remain more efficient for OLTP-style executions.

Another study by Salunke & Ouda (2024) compares PostgreSQL and MySQL in the context of continuous user authentication systems. Using a Python-based benchmarking framework, the authors conclude that PostgreSQL is better suited for such systems due to its superior performance in handling SELECT operations (both conditional and non-conditional) and its robust concurrency control capabilities.

Reflecting technological advances, some studies evaluate databases deployed on cloud platforms. For example, Wiechork & Charão (2020) assesses Moodle, a social learning management framework compatible with PostgreSQL, MySQL, MariaDB, and other SQL systems. The study highlights the importance of selecting cloud service providers based on performance and billing plans. Leveraging Infrastructure as a Service (IaaS), the authors use benchmarks like *pgbench* and *sysbench* to measure performance across cloud instances. The main finding is that Amazon Web Services (AWS) instances consistently outperform Google Cloud Platform (GCP) instances. *Pgbench*, which we also utilized in our study, is the built-in benchmarking tool for PostgreSQL. *Sysbench*, a versatile benchmarking tool, was used to evaluate database performance along with metrics such as file I/O, memory allocation, and thread scheduling.

Several similar studies compare database performance under read, write, scan, and update operations (Filip & Čegan, 2020; Tongkaw & Tongkaw, 2016; Bog et al., 2008). These works propose methodologies for analyzing performance under both transactional and non-transactional workloads, emphasizing metrics such as throughput, average latency, and resource utilization (CPU/memory) under varying workload intensities. Moreover, when concurrency is the focus, they apply different levels of threads and workers to observe resource utilization in terms of CPU and memory consumption for OLTP-style workloads. While these studies rarely delve into system internals, they provide frameworks for meaningful comparisons. Our study differentiates itself from the aforementioned studies by focusing on concurrency control principles and trends in modern database systems. Specifically, we analyze implementation strategies in PostgreSQL and evaluate them using its built-in benchmarking tool, *pgbench*.

The second group of related work investigates the impact of concurrency protocols on system performance. The study by Ports & Grittner (2012) on the SSI (Serializable Snapshot Isolation) implementation in PostgreSQL demonstrated that SSI achieves performance slightly below Snapshot Isolation (SI) but significantly higher than traditional two-phase locking (2PL), particularly under read-intensive workloads.

Zendaoui & Hidouci (2015) explored the intrinsic factors influencing SSI's performance in PostgreSQL. Their experiments used a custom benchmark, SIBenchmark, designed to simulate read-write conflicts between read-only and update transactions. Key parameters varied included:

- Isolation levels (SI vs. SSI),
- Multiprogramming level (number of concurrent threads),
- Transaction mix (read/update ratios, ranging from pure updates to pure reads),
- Transaction weight (number of rows selected or updated per transaction).

Key performance metrics were throughput, abort rates, and transaction execution times. Their findings revealed that SSI aborts more transactions than SI due to additional checks for dangerous structures (patterns in the serialization graph that risk non-serializable outcomes). However, these structures had minimal impact on overall benchmark performance. Despite higher abort rates, SSI maintained throughput comparable to SI, showing resilience to performance degradation.

Our study aligns with Zendaoui & Hidouci (2015) but extends the analysis by:

- Using PostgreSQL's built-in *pgbench* tool to evaluate all isolation levels (Read Committed, SI, and SSI),
- Assessing memory consumption across isolation levels,
- Investigating how database scalability impacts isolation-level performance.

3. BASIC CONCEPTS AND CURRENT TRENDS

Transaction isolation levels are defined in the ANSI SQL in four levels: Serializable (S), Repeatable Read (RR), Read Committed (RC), and Read Uncommitted (RU) (ANSI, 1992). Anomalies occur depending on the degree of isolation. The well-known errors are named dirty read/write, lost update, non-repeatable read, skewed read/write, and phantom read/write. Today, many relational database systems are based on the foundational principles introduced by the System R model in the mid-1970s. This model still lays the groundwork for some modern implementations of isolation levels and transaction recovery mechanisms as it provides strong correctness for high-contention workloads (e.g., banking). On the other hand, Snapshot Isolation (SI), first conceptualized as part of the Multi-Version Concurrency Control (MVCC) model in the early 1980s (Bernstein & Goodman, 1981) and further detailed in research during the mid-1990s (Berenson et al, 1995), has recently become

prevalent in many modern database systems. A significant recent development in this domain has been the introduction of Serializable Snapshot Isolation (SSI), formally defined in 2008 along with its successful integration into PostgreSQL's MVCC framework in 2012 (Cahill et al, 2008) and (Ports & Grittner, 2012). SSI is gaining traction in centralized systems and is a promising candidate for broader adoption, though its integration into distributed systems is still an active area of research and has not yet become the standard (Kleppmann, 2017), (Cao et al, 2022) and (Binnig et al, 2014).

Despite inconsistencies in the taxonomy, conflicting definitions have arisen due to the fact that Snapshot Isolation (SI) and Serializable Snapshot Isolation (SSI) are not part of the current SQL standard. Notable conflicting definitions will be highlighted. Neither PgSQL nor Oracle implement Read Uncommitted (RU) isolation. In IBM Db2, Read Committed (RC), the default isolation level, is referred to as Cursor Stability (CS). Additionally, IBM Db2 provides Serializable isolation through Repeatable Read (RR), implemented with 2-Phase Locking (2PL). In contrast, Oracle presents the Serializable isolation level using Snapshot Isolation (SI) under the same name. In SQL Server, along with the four standard isolation levels, Snapshot Isolation (SI) is defined separately. In PgSQL, Repeatable Read (RR) is implemented using a variant of Snapshot Isolation, but it is distinct from Serializable, which uses Serializable Snapshot Isolation (SSI). A more comprehensive comparison, though somewhat dated, is available (URL-1).

3.1. Serializable Plans and Concurrency Methods Generating Them

A serializable plan means that concurrent transactions run as if they correspond to any serial execution. Since the sequence of "read-write" or "write-write" operations from different transactions on shared data changes the result, it is named a conflict. Serializability can be achieved by producing concurrent plans in which the order of processing of all conflicts is the same. This is essentially a strict constraint on accuracy, and there are some other definitions, such as view-based and debit-credit, which can be defined under some conditions (Silberschatz et al, 2019). Today, many centralized database systems are based on conflict-based serializability and provide this with different methods: 2-Phase Lock (2PL), Timestamp-Order (TS/O), Optimistic Concurrency Control (OCC), and Multi-version Concurrency Control (MVCC). The basic principles of these methods are explained in detail in many database textbooks (Silberschatz et al, 2019). In summary, the 2PL protocol is a lock-based pessimistic approach, arranging conflicts through collecting and releasing locks in two distinct phases. In the TS/O method, conflicts occurred regarding the start time of the transactions are executed; otherwise, the transaction is immediately aborted. In the optimistic approach, transactions first run independently in their local memory, and after verification, the write actions are atomically saved to the disk. Transactions run simultaneously and very quickly without any delays other than the access delay for the data from the disk, and the transactions that fail to pass the verification at the end are aborted. Finally, in the MVCC model, the basic idea is to save old data versions (usually the record) at each update. Thus, each transaction is allowed to see the appropriate record version (database snapshot) required for serializability without using any locks. While this idea serves as the foundation for Snapshot Isolation (SI), it can lead to some complex and challenging anomalies in its simplest form. For this reason, there have been efforts to guarantee serializability within the MVCC structure using lightweight algorithms, leading to Serializable SI. MVCC is not only for SI; it can also serve as an infrastructure for other methods, such as 2PL, TS/O, and OCC (Silberschatz et al, 2019).

The impact of each approach on system performance varies. For example, in lock-based methods, long lock waits, lock table management, and (though rare) dead-lock-induced rollbacks are the most significant adverse effects. In TS/O, recording the timestamp for each disk operation and the fact that some transactions are subject to aborting are concerning for system throughput. The burden of OCC on the system is to collect read and write information during the execution and to abort transactions that fail to pass verification even if the transaction has been completed. In the MVCC infrastructure model, the primary burden is the space loss caused by multiple versions. Additionally, the vacuuming process—a system routine that periodically releases space occupied by old versions—adds to this overhead. Moreover, in Snapshot Isolation (SI) and specifically in Serializable Snapshot Isolation (SSI) implementations, another critical concern is the overhead associated with tracking write-read, write-write dependencies, and rw-antidependencies necessary for detecting anomalies (Ports & Grittner, 2012). We will define this concept in more detail in Section 4.2.

3.2. Usage Examples of Concurrency Methods in Database Systems

The traditional method for implementing concurrency and isolation levels is through lock protocols, as seen in widely used systems like MySQL, MSSQL Server, and DB2. While the 2PL protocol ensures serializability and addresses complex isolation issues like skewed writes and lost updates, its heavy system load often leads to a preference for alternative methods. Recently, 2PL has primarily been employed in a supporting role alongside MVCC or OCC frameworks in modern system designs for three main reasons.

First, 2PL transactions maintain write locks until the commit point to prevent cascading rollbacks, which can increase the likelihood of deadlocks and reduce concurrency. While more complex 2PL methods that allow early lock releases are possible, they require advance knowledge of transaction operations, complicating implementation.

Second, in hybrid workloads, long-term reporting transactions can be impeded by read locks, while write locks from short-term operational transactions may delay read-only operations (Berenson et al, 1995).

Lastly, the stateless nature of today's web applications complicates the use of 2PL (Silberschatz et al, 2019). For instance, using 2PL for a seat reservation operation, which includes several sub-operations (like seat status checks, user selections, confirmations, and payments), leads to inefficient lock usage. Instead, each sub-operation can be treated as a separate HTTP connection and transaction, reducing user input delays but risking non-serializable schedules due to concurrent updates. To mitigate this, applications can verify updates from other clients by comparing them with the most recently read values, such as checking if a selected seat is still available. If verification fails, the selection process restarts. This approach, while fundamentally optimistic, offers flexibility at the application level and operates independently of the database's concurrency model.

Despite these disadvantages, the 2PL model remains relevant in many databases; it often coexists with other methods. For example, in PgSQL, metadata operations adhere to 2PL, while only exclusive locks are used for data updates within the MVCC framework.

We have not come across an example of a system where the TS/O method is applied alone. It embodies an optimistic approach, allowing transactions to start running immediately without the need for lock waiting. It distinguishes itself from OCC in that non-serializability is detected earlier while the transaction is still active, rather than at the end of the transaction.

Several relational and NoSQL databases, including Redis and CouchDB, utilize the Optimistic Concurrency Control (OCC) model. This model extends beyond the database level and can be applied at the application level, as illustrated by the earlier reservation example. OCC is particularly efficient in scenarios with fewer transaction conflicts, such as report-intensive systems like MonetDB, which rely exclusively on this model. However, in operational applications with numerous conflicting operations, the efficiency of OCC may decrease due to a higher rate of aborts. In these cases, a cautious approach using 2PL or other pessimistic methods at lower isolation levels may be more suitable. Beyond databases, optimistic methods are prevalent in managing update conflicts in shared object repositories, such as Hibernate and wiki pages. Additionally, optimistic approaches are widely used in modern systems that require simultaneous access to objects in main memory, particularly in bridge architectures like Hibernate and JPA (Java Persistence Architecture), which efficiently abstract database access for application developers.

3.3. Alternative Approaches to Being Serializable

In the previous section, we discussed the approaches based on the idea of serializability. While current methods generally align with these principles, advancements in system memory capacity have strengthened the feasibility of actual serial execution of transactions. High-capacity memory can hold entire active datasets, and the short latency of operational transactions suggests that running all transactions serially on a single thread may be sufficient.

Moreover, serial execution can enhance system performance by eliminating the overhead associated with concurrency algorithms, such as lock waiting and transaction aborts. However, achieving high performance requires some restrictions. User-interactive multi-query transactions in web applications often involve user-related waits and network delays. Thus, high performance can only be realized if an entire transaction executes on the server within the scope of a stored procedure, with the active dataset held in memory.

Stored procedures defined by SQL/PSM in the SQL 1999 standard have fallen out of favor and are often difficult to use. However, some modern systems are reviving stored procedures by allowing the use of general-purpose programming languages for transaction definitions. For instance, "VoltDB" and "Datomic" databases enable transactions to be defined using Java, while "Redis" supports Lua scripting for this purpose.

4. CONCURRENCY PROTOCOL IN PGSQL

As of version 9.1 in 2011, PostgreSQL (PgSQL) implements serializability optimistically through Serializable Snapshot Isolation (SSI). It also supports weaker isolation levels using its MVCC (Multi-Version Concurrency Control) infrastructure. In this model, transactions can read versions permitted by the isolation level without acquiring read locks, while write operations require exclusive locks. Update conflicts are detected during the verification stage at the commit point, resulting in either successful or unsuccessful transaction completion based on the isolation level. PgSQL's concurrency model reduces wait times but may lead to occasional aborts due to non-serializability issues. Unlike 2PL, reads and writes do not wait for each other on shared data; only write locks wait. Additionally, unlike OCC, active transactions

operate directly on shared table pages in the buffer pool, rather than copying tables or pages to a local workspace.

4.1. MVCC in PgSQL

In PostgreSQL, the MVCC model is implemented using transaction IDs, following three key principles to determine the visible versions for transactions:

1. Transaction IDs start from 1 and always increase in ascending order (0 is an invalid transaction number).
2. The visibility of a record version is determined by two system-level attributes, tx_min and tx_max. These attributes indicate the transaction that generated the record version and the one that rendered it obsolete, respectively.
3. A record version becomes visible only after the transaction that generated it has committed. The commit status of each transaction is tracked in a commit log bit array in main memory, indexed by the transaction ID.

Accordingly, the last version of a record visible to other transactions is the one where tx_max = 0 and the corresponding tx_min transaction has committed. Previous versions are visible to transactions with IDs between the tx_min and tx_max values.

This concept can be illustrated with a simple example, as seen in Figure 1. Let Ti represent transaction i. Figure 1 shows the state of the record versions and the commit log bit array (clog array) during the moments when T99 performs the read operation R(X). In the first read of R(X), the value X0 will be read, since this is the last update made by transaction T80, and the clog array indicates that T80 has committed.

On its second read, T99 will see the record version according to its isolation level. For instance, if T99 operates under Read Committed (RC), it will read version X1, as this is the last update made by transaction T100 , and the clog array indicates that T100 has committed. However, under Repeatable Read (RR), it will see version X0 again, which is visible at the transaction's start point.

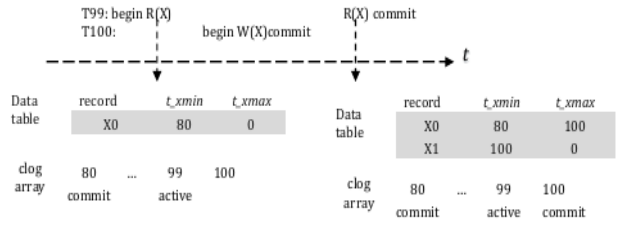


Figure 1. An update and related visibility in the MVCC model

While there are simpler structures in MVCC, it also encompasses more complex concepts, such as tx_snapshot. The system becomes intricate, involving considerations such as aborting and committing transactions, updating or deleting data records multiple times within the same transaction, managing subtransactions, addressing the potential exhaustion of transaction IDs, and handling indexes that point to different versions. For readers interested in exploring these details further, we recommend consulting sources (URL-2) and (URL-3).

Another challenge posed by the MVCC structure is the rapid increase in data volume due to new versions. To manage this, the system periodically deletes versions that are no longer visible, accepting some level of space loss.

4.2. Implementation of Isolation Levels in PgSQL

As discussed in previous sections, transactions often operate with weaker isolation levels to mitigate the negative impact of strict serializability on system throughput, which can lead to anomalies in data access races. We will not define well-known anomalies such as dirty reads, non-repeatable reads, and phantom reads, but will instead focus on the isolation levels in PostgreSQL. Then we will briefly discuss more obscure anomalies incurred by Snapshot Isolation and how they are handled in the Serializable SI.

The default isolation level in most systems, including PostgreSQL, is Read Committed (RC), which provides sufficient isolation, especially for web applications. Conflicts are often resolved at the application level. For example, in an airplane seat reservation application, a common non-repeatable read anomaly occurs when a user selects a seat that becomes unavailable later. This integrity error is easily detectable and can be addressed by allowing the user to choose a different seat. However, higher isolation levels may negatively impact service quality by slowing down booking processes.

Snapshot Isolation (SI), also known as Read Repeatable (RR) Isolation in PgSQL, eliminates non-repeatable reads, lost updates, and phantom errors because a database transaction only sees the records that were committed at the time it started. This means that any insertions, deletions, or updates occurring after the transaction begins will not be visible to it, except for the versions generated by the transaction itself. Unfortunately, other errors, such as skewed writes and potentially more complex issues, remain possible (Ports & Grittner, 2012). SI isolation is most suitable for long-term read-only transactions, such as reporting (i.e., OLAP transactions) and backup operations. SI isolation allows long-term transactions to see identical data versions for every read until the end (i.e., the versions that were valid at the start time). SI isolation's overhead on the system is higher

compared to RC (Read Committed), primarily due to the need to keep many possible versions of records for extended periods. On the contrary, assuming all transactions are in RC isolation, only two versions of each updated data record would be sufficient for all times (Kleppmann, 2017). The processes of detecting write errors and aborting may adversely impact SI's performance, particularly under OLTP-heavy workloads. In contrast, under RC, aborts are rare occurrences triggered solely by deadlocks.

In Serializable Snapshot Isolation (SSI) in PostgreSQL, the expectation is that no anomalies will occur, as it is designed to prevent various concurrency issues. While reads are consistent with those in Serializable Isolation (SI), all update-related conflicts are detected at the end of the transaction, during the commit phase. Anomalies that can be detected in SSI but not in weaker isolation levels include skewed writes and other subtler anomalies including read-only transactions. Now, let us define skewed writes and discuss how this error is detected using intention locks in PostgreSQL.

The skewed write issue arises when transactions violate an integrity constraint by performing update or insert operations on different sections of the same dataset. Transactions often write (update or add) records based on certain conditions, inadvertently altering the integrity constraint with these actions. This problem is essentially a consequence of transactions in Snapshot isolation seeing versions only at their start time and being unaware of changes to integrity conditions.

For instance, consider a meeting room reservation application where the same room should not be allocated to different users during the same period. In SI isolation, two concurrent transactions may perceive a room as available at their start time, generate and insert reservation records simultaneously, and complete their transactions. This situation can easily lead to a reservation that violates the integrity constraint (Kleppmann, 2017).

To elaborate on the skewed write problem, we will revisit the shift organization problem detailed in references (Kleppmann, 2017) and (Ports & Grittner, 2012). Unlike the previous reservation example, the integrity constraint in shift organization changes by updating existing records rather than inserting new ones. For instance, suppose the "shift" table (name, shift_status, shift_no) maintains the duty status of doctors. In this scenario, the integrity constraint asserts that there should be at least one doctor on duty during any given shift.

Consider a situation where an application module allows doctors to relinquish their shifts for various reasons. Figure 2 illustrates this application module. In our example, assume that shift 123 has only two doctors assigned, and they both attempt to give up their assignments simultaneously. When both processes start at approximately the same time, they can easily set the current_number_of_on_shift to 2 and then change the shift_status to false, thus violating the integrity constraint.

```
begin tx;
current_number_of_on-shift <-
        select count(*) from shift where shift_status ='true' AND shift_no=123;
if(current_number_of_on-shift >=2)
        update shift set shift_status ='false' where name = :name AND shift_no=123;
end tx;
```

Figure 2. Shift relinquish action

As evident from the examples above, the skewed write problem is associated with read-write conflicts, meaning that the schedule does not correspond to any serial plan. Since these read-write conflicts are not ordinary and include anomalies such as lost updates and phantom records, they represent an anomaly that cannot even be detected using the traditional Strict 2PL protocol. In the first example (meeting room reservation), the integrity constraint is violated because a phantom record is added. In the second example, transactions can be completed without satisfying the condition because writing operations are performed on different values.

For more complex examples of the skewed write problem involving more than two transactions and for the read-only transaction anomaly, we recommend reference (URL-3) for more curious readers. These are more interesting problems based on read-write conflicts caused by read-only transactions, unlike the skewed write problem. In 2004, Fekete et al. demonstrated that Snapshot Isolation's (SI) core principle—read-only transactions seamlessly coexisting with writes—is highly unlikely (Fekete et al,2004). This realization sparked research into Serializable SI. Now, we will briefly touch on the definitions and related theorems used in SSI theory, and then we will discuss some practical methods employed in the SSI implementation in PgSQL.

In Serializable Snapshot Isolation (SSI), which is type of optimistic concurrency control method, a final verification of the transaction is necessary. This verification involves detecting conflicts in a precedence graph known as the multi-version serialization graph (MVS-graph). The MVS-graph consists of a node for each transaction, with edges connecting some nodes to represent dependencies and anti-dependencies. Two types of dependencies—write-read (wr) and write-write (ww)—and one anti-dependency, read-write (rw), were originally proposed to represent the MVS-graph in the study by Adya et al. in 2000 (Adya et al., 2000). These concepts later served as a baseline for the SSI proposal in 2008 and its implementation in PgSQL in 2012 (Cahill et al., 2008; Ports & Grittner, 2012).

In the context of dependency, it refers to a situation where one operation sees the write of a

previous operation (wr dependency) or when one operation overwrites the write of a previous operation (ww dependency). On the other hand, in the case of antidependency, it means that one operation does not see the write made by another operation (rw antidependency). If a cycle exists in the MVS-graph, then the execution does not align with any serial order, indicating that a snapshot isolation anomaly has resulted in a serializability violation. For example, in the shift relinquish scenario previously discussed, one doctor updates the row containing his call status, but this update is not visible to others' SELECT queries, creating a read-write antidependency. Similarly, another doctor's UPDATE is not visible to the first doctor, resulting in a second read-write antidependency in the opposite direction. The resulting cycle indicates that the execution is not serializable.

Detecting ordinary lost updates caused by ww dependencies is straightforward in Serializable Snapshot Isolation (SSI) through the use of exclusive locks, rather than by detecting cycles. SSI employs a **'first-commit-wins (FCW)'** policy, which contrasts with SI's **'first-update-wins(FUW)'** rule. However, detecting subtler anomalies requires identifying cycles in the MVS-graph, which imposes significant overhead on the system. To address this, lightweight algorithms (e.g., tracking specific dependencies instead of full-cycle detection) are often used in SSI implementations. For example, the concept of a "dangerous structure" is defined as two consecutive rw anti-dependency arcs in the MVS-graph, serving as an indicator of a possible cycle in the graph. The related theorem and its corollaries have been proven (Cahill et al, 2008). Hence, the problem is reduced to detecting these dangerous structures, which is significantly easier than identifying a cycle within the entire MVS-graph. It is important to note that a dangerous structure is a necessary, but not sufficient, condition for the presence of an anomaly, which can lead to false positives. Here, the SSI implementation in PgSQL is based on these dangerous structures and uses a lightweight lock called **SIRead locks** to track them dynamically at runtime. **SIRead locks** (pseudo-read locks) efficiently track **rw-antidependencies** without requiring exhaustive graph traversal (Ports & Grittner 2012). SIRead lock differs from traditional read locks that block writes; instead, it tracks dependencies between transactions, further reducing computational costs.

However, in some cases, this (anti)dependency information may need to be stored even after the transaction has been committed. For example, Figure 3 illustrates a transaction Ti with its start timestamp (startTS(Ti)) and commit timestamp (endTS(Ti)), and shows how Ti may conflict with other transactions like Tk or Tm. The system checks for conflicts with any transactions that overlap with Ti. In the event of a ww conflict with Tk, Ti will abort based on the "first-commit-wins" rule to prevent a lost update. Similarly, a rw conflict, detected via the SIRead lock, may also cause Ti to fail.

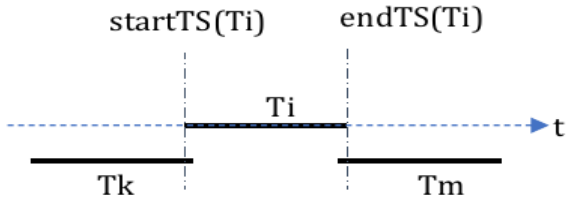


Figure 3. Verification of Ti at commit time

Preserving both intention locks (i.e., the read set) and write information (i.e., the write set) even after a transaction has completed may be a critical concern for long-running OLAP transactions. As illustrated in Figure 3, the use of intention locks by transaction **Tk** at the commit time of **Ti** highlights this challenge. However, recent versions of *PostgreSQL* suggest that under certain conditions, serializability can be detected earlier (URL-7). For example, if **Tk** has already committed, **Ti** may fail and abort without needing to wait until its own commit point to detect a conflict.

Another limitation of SSI is that it can only function if all other active transactions are also operating under SSI, since only transactions in *Serializable isolation* can utilize intention locks (Kleppmann, 2017).

A further concern is the occurrence of **false non-serializable decisions** (URL-2), where transactions may be unnecessarily aborted. As previously stated, since anomaly detection in the multi-version serialization graph is performed using heuristics instead of complete cycle detection, false positives and unnecessary aborts are inevitable. Furthermore, this issue often stems from internal factors, particularly related to the query execution plans chosen during transaction processing. For instance, if the query optimizer selects a heap scan instead of an index access, the **SIRead lock** might be applied at a coarser granularity—such as at the page or file level. In these cases, even when there is no direct conflict between the write set of one transaction and the read set of another, the broader exclusive lock and the coarser SIRead lock may introduce artificial conflicts in the dependency detection graph. The selection of lock granularity is thus crucial. It requires balancing performance and accuracy in transaction management algorithms, a principle that also applies to the implementation of SSI (URL-3).

Despite the concerns mentioned briefly in the previous paragraphs, experimental studies have suggested that the burden may not be as severe as that observed with Two-Phase Locking (2PL) (Ports & Grittner, 2012).

Nevertheless, the general acceptance of Snapshot Isolation (SI) and Serializable Snapshot Isolation (SSI), along with their fulfillment of many promises, does not eliminate the relevance of

weaker isolation levels and other traditional solutions (i.e. application-level solutions) in today's database-driven applications (Beillahi et. al., 2021) and (Alomari et. al., 2014). First, it might be more efficient to execute transactions in weaker isolation levels, while using queries in the form of SELECT FOR UPDATE. In some cases, write skew can be avoided by locking all records read in the selection clause with FOR UPDATE. However, this query-level approach should be applied cautiously to prevent not only write skew but also other anomalies, depending on the database access patterns of the application (Kleppmann, 2017).

Secondly, the *lost update* problem in *Read Committed* (RC) isolation can be more effectively prevented by using atomic writes (if supported), special atomic operations such as **compare-and-set (CAS)**, or by implementing explicit locks at the application level (Kleppmann, 2017).

Last but not least, a notable study by Wang et. al.(2016) introduces the serial safety net (SSN), an efficient and general-purpose certifier aimed at enforcing serializability across various concurrency control (CC) schemes, including read committed (RC) and snapshot isolation (SI). This method serves as an external solution to the problem, as SSN does not dictate access patterns—this responsibility lies with the underlying CC scheme. Rather, it focuses on tracking dependencies and aborting transactions that could potentially close a dependency cycle if permitted to commit.

5. EXPERIMENTAL RESULTS

Comparative test environments are essential for measuring database performance metrics, including average transaction latency and throughput (typically expressed as transactions per second, TPS), as well as resource utilization. To address this need, various benchmark software tools are available, allowing for a comparison of specified criteria across different database systems (URL-4), (URL-5). In this study, we utilized the widely employed performance evaluation software known as *pgbench* (URL-6) within the PgSQL ecosystem. The *pgbench* module serves as a benchmarking tool, enabling the evaluation of PostgreSQL's performance under various workloads. While PostgreSQL has nearly 400 configuration parameters, *pgbench* specifically helps assess performance impacts related to throughput and latency.

pgbench simulates a real-world scenario where multiple clients connect to the PostgreSQL system and execute transactions concurrently through each connection. Key input parameters include the workload definition, client-server connection setup, the number of concurrent threads for multi-CPU environments, and the database scale.

Workload: *pgbench* uses a TPC-B-like (URL-7) schema by default, consisting of four tables: **Accounts, Branches, Tellers,** and **History**. Transactions involve standard SQL operations such as deposits, withdrawals, and balance queries. There are four transaction types: **tpc-b (default), no-branch, update-only,** and **select-only** transactions. In the **tpc-b** workload, deposit and withdrawal transactions are simulated through a sequence of five SQL operations: 1) deposit/withdraw a random amount to/from a random account, 2) read the account's final balance, 3) update the teller's balance, 4) update the total turnover of the branch, and 5) insert a record of the transaction into the history table. Steps 4 and 5 experience high contention due to updates on smaller tables (e.g., the default configuration has 10 tellers and 1 branch). Other existing transactions involve different combinations; for example, **select-only** queries the balance of a random account exclusively, **no-branch** omits updates on smaller tables (i.e. **Branches, Tellers**), and **update-only** performs deposit/withdrawal operations solely on the accounts table. Additionally, *pgbench* supports custom workloads using user-defined schemas. In this study, we always base our workloads on the default schema and conduct performance measurements using built-in transactions in some experiments. Additionally, we use our defined transactions in Experiments 3 and 4 to induce subtler anomalies, such as skewed reads or writes. For readers who wish to understand the experiments in more detail, we recommend consulting the PgSQL documentation for information on the mentioned schema, the types of built-in transactions, and other configuration settings of *pgbench*.

Connection configuration: In its default mode, *pgbench* initiates a session for each client, and multiple transactions are transmitted to the system within each session. By default, PostgreSQL allows up to 100 concurrent clients, though this limit can be adjusted based on system needs.

Determining the number of concurrent threads in multi-core or multi-cpu environments: *pgbench* allows workloads to be distributed across multiple threads in multi-core or multi-CPU environments. To optimize performance, the number of available CPU cores should be considered when determining the number of concurrent clients. For example, on an 8-core machine, running 8 clients would result in each core handling one active thread. If there are 9 clients on an 8-core machine, the system will distribute the threads as evenly as possible across the cores, with one core handling an extra client. To ensure optimal performance in our experiments, we selected the number of clients (4, 8, 16, etc.) to match the number of available CPU cores.

Database scale: *pgbench* simplifies the evaluation of system performance scalability. At the default “scale factor” of 1, the tables initially contain 1 branch, 10 tellers, 10000 accounts and 0 history records. System throughput, measured by the Transactions Per Second (TPS) rate, can vary significantly based on how much of the database fits

into available memory. The size of the database and the amount of allocated memory are key factors in selecting the appropriate indexes and tuning recovery settings. For our experiments, we configured 2GB of shared memory (shared_buffers), which equates to 25% of the total available memory, following guidelines from (URL-8) and other sources. To focus on the balance between performance and isolation level accuracy, we used datasets that fit into main memory for most tests. Only in our final experiment did we test with larger datasets (2GB and beyond).

The primary outcome of the *pgbench* experiment is the transaction rate and the corresponding average transaction termination time. Given that concurrent execution leads to a linear increase in the TPS rate, the relationship with latency can be expressed as Equation 1. Let t_a (msec) be the average transaction termination time and c be the number of clients,

$$TPS = (1000/t_a) * c \quad (1)$$

Additionally, the rate of aborted transactions in RC (due to deadlock) and higher-level isolations such as SI and SSI (due to non-serializability) is reported. Optionally, the number of re-runs of aborted transactions is also documented. The default setting is that failed transactions are not retried. In our experiments, we opted to repeat unsuccessful transactions until they were successful (or until the end of the test).

We conducted our *pgbench* experiments on a machine equipped with a 4-core CPU (i7-7700), 8GB RAM, and SATA SSD hardware. The PostgreSQL system configuration settings used are as follows:

PostgreSQL ver.: PostgreSQL 15.4 (Ubuntu 15.4-1.pgdg22.04+1) on x86_64-pc-linux-gnu

shared memory (i.e., shared_buffers) = 2GB (25% of our available memory)

AUTOVACUUMING=off, FILL FACTOR = 75, WAL_LEVEL= minimal, MAX_WAL_SIZE=2GB, SYNCRONOUS_COMMIT=off

We configured AUTOVACUUM to be off because, during our tests, we perform vacuuming, analysis, and checkpoints, subsequently running tests on "cold" memory (cold cache) when loading a new database. The FILL FACTOR parameter pertains to the occupancy rate of heap and index pages. While *pgbench* defaults to generating tables with 100% occupancy, we opted for a value of 75% to provide space on the same page for new versions generated by the MVCC infrastructure. Ultimately, with these adjustments we aimed to reduce the recovery load on system throughput. For readers interested in recovery-based settings, we recommend consulting PostgreSQL's Documentation.

We crafted the scenarios and visualizations with the necessary adjustments and script file refactoring in the (URL-9) project. To ensure consistency and reproducibility of the graphs, we established a "common state" by conducting 100-second tests, iterating multiple times. For each test, the database is reloaded and runs on clean memory.

5.1. Experiment 1

In this experiment, our objective is to illustrate the key characteristics of the current workloads in *pgbench* and to highlight the impacts of the configuration decisions mentioned in the previous paragraphs. We consecutively execute four different workloads, each involving 8 clients, for 100 seconds on a 400 MB database. The isolation levels are consistently set to Read Committed (RC) across all scenarios. Since we achieved the expected results for the TPS and latency plots, we have preserved them for use in subsequent experiments.

Figure 4 depicts the TPS rate for the workloads described earlier. In general, the graphs align with our expectations. For example, the TPS curve resulting from approximately 7 million transactions, with an average completion time of 0.117 milliseconds in select-only tests, appears remarkably consistent.

As the dataset fits entirely into main memory, there is no delay from Write-Ahead Logging (WAL) due to updates, and no victim page selection is required. However, in workloads involving updates, the noticeable jitter in the TPS curve is caused by WAL writes. Our goal with the new configuration was to speed up the loading of test tables without generating log records and to reduce TPS jitter by suppressing WAL writes in update transactions.

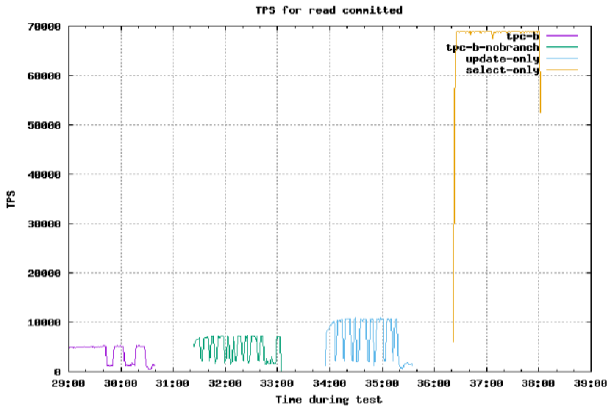


Figure 4. TPS rate with default configuration

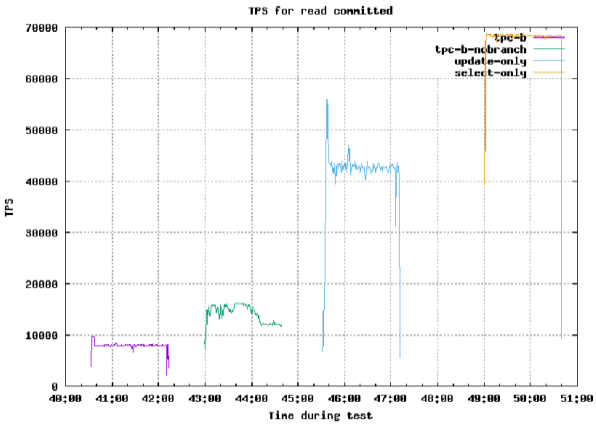


Figure 5. TPS rate with revised configuration: synchronous_commit=off, wal_level=minimal

Figure 5 demonstrates that the new configuration is most effective in update-only scenarios, while it provides no noticeable advantage in select-only cases.

5.2. Experiment 2

In this experiment, our objective is to demonstrate the favorable performance of the default RC isolation level for typical Online Transaction Processing (OLTP)-style workloads. To this end, we executed the **tpc-b** workload with 8 clients, each utilizing eight threads, on a database of approximately 160 MB. Since the database includes ten branches by default, and the **tpc-b** transaction involves withdrawal and deposit operations per each branch, we set up 8 clients to ensure an acceptable level of contention during updates.

Figure 6 presents the TPS throughput and Figure 7 presents memory usage graphs obtained from the 100-second tests, each conducted at different isolation levels.

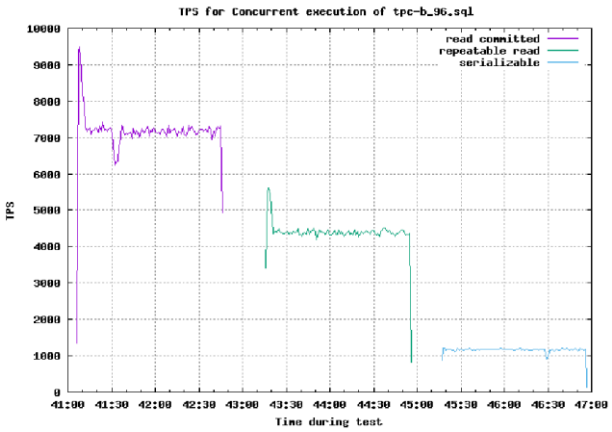


Figure 6. Outcomes of tpc-b workload: system throughput

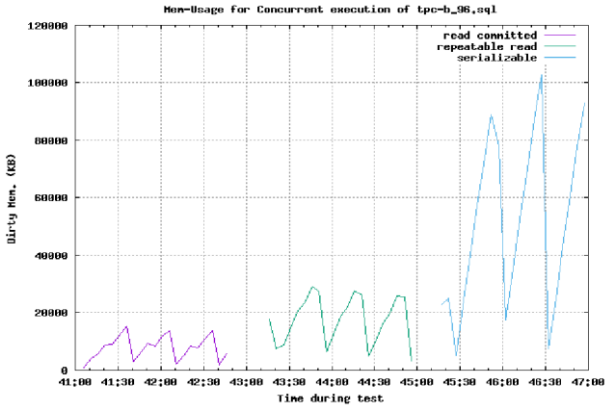


Figure 7. Outcomes of tpc-b workload: memory consumption

In the Repeatable Read (RR, or SI) configuration, approximately 40% of all transactions, and only 18% in Serializable Snapshot Isolation (SSI), required reruns. In contrast, under RC isolation, where serializability is not a concern, no deadlocks were encountered. The reduced TPS in SI and SSI primarily results from the overhead of handling unsuccessful transactions and managing write-write and read-write dependencies. While the failure rate is lower in SSI compared to SI, the more intricate conflict tracking in SSI negatively impacts performance. As defined in the beginning of Section 5, the **tpc-b** test experiences high write contention during the 3rd and 4th steps (due to updates on small tables) and very few read-write dependencies in the 1st and 2nd steps (since the account is a large table). Therefore, we can conclude that SI handles frequent write-write conflicts better than SSI.

To substantiate this interpretation, we monitored memory usage during the tests at 5-second intervals (using the /proc/meminfo file). On average, memory usage is about two times higher in SI compared to RC, while it can escalate to approximately four times higher in SSI compared to RC.

5.3. Experiment 3

In this experiment, our goal is to illustrate the performance of Snapshot Isolation (SI) in a workload involving skewed reads. To achieve this, we defined transactions tx1 and tx2 as depicted in Figure 8. The first half of the **Accounts** consists of checking accounts with 0 liras each, while the other half consists of investment accounts with 100 liras each. The database scale is 1, meaning we have a single branch with 100,000 accounts, which translates to 50,000 customers each having two accounts.

```
\set aid random(1, 100)
\set aid_offset (100000 * :scale /2)
BEGIN;

SELECT abalance AS b1 FROM accounts
WHERE aid=:aid \gset

SELECT abalance AS b2 FROM accounts
WHERE aid=:aid+ :aid_offset \gset

\SET t :b1 + :b2
\IF  :t != 100
        \shell echo " skew read error!!! " :t
\ENDIF
END;
```

```
\set aid random(1, 100)
\SET aid_offset (100000 * :scale /2)
\set delta 100
BEGIN;

        UPDATE accounts
        SET abalance = abalance - :delta
        WHERE aid =:aid+ :aid_offset;

        UPDATE accounts
        SET abalance = abalance + :delta
        WHERE aid = :aid;

END;
```

Figure 8. Transactions for Experiment 3

Tx1 retrieves the total amount of money for a customer, and tx2 transfers money from the customer's investment account to their checking account. When these transactions are run together, data integrity is maintained if tx1 always sees a total of 100 liras. However, if a transfer occurs for the same customer between tx1's steps of setting b1 and b2, the total seen by the customer may be incorrect, potentially leading to a skewed read anomaly, which may occur under Read Committed (RC) isolation. Note that tx1 never interleaves between the two updates of tx2, even at the RC isolation level.

To increase the likelihood of this skewed read error, we executed tx2 500 times more frequently than tx1, concentrating the transfers within the first 100 accounts by setting aid to 100. Figure 9 shows the TPS and Figure 10 shows memory usage graphs over time for tests run for 100 seconds at different isolation levels.

On average, the performance sees a marginal decrease of only 3% under SI (compared to RC), and a 5% decrease under Serializable Snapshot Isolation (SSI) relative to RC. Although memory consumption slightly increases in SI and SSI, the RC test revealed that 60 out of 2,843 tx1 transactions resulted in incorrect calculations, leading to an error rate of approximately 2%. In contrast, under SI and SSI isolation, around 4% of the total transactions were aborted to resolve conflicts and prevent anomalies.

Under SI isolation, reads never conflict with writes. Consequently, tx1 reads the values of b1 and b2 from a database snapshot taken at the beginning of the transaction, ensuring that the total calculation is accurate. The failures observed in SI and SSI arise primarily from write-write conflicts between concurrent tx2 transactions. Despite these failures, SI isolation maintains high throughput, experiencing minimal performance loss compared to RC, while ensuring correctness. Similarly, the advantages of SI isolation become more evident in workloads involving long OLAP-style transactions and short OLTP-style transactions.

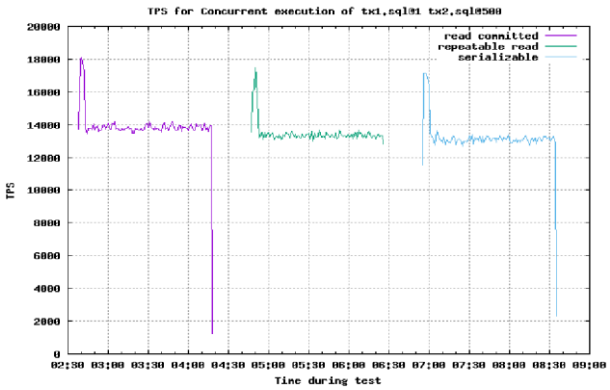


Figure 9. Outcomes of workloads containing skewed reads: system throughput

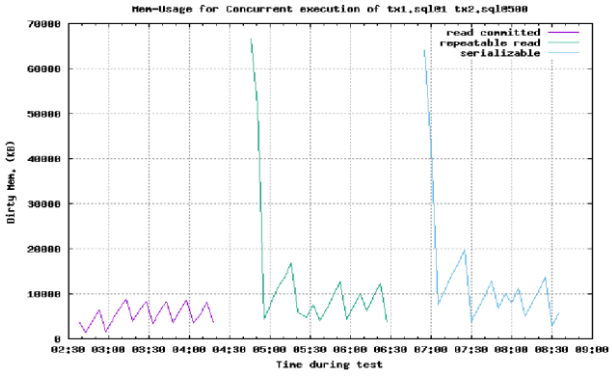


Figure 10. Outcomes of workloads containing skewed reads: memory consumption

Another significant metric is the table bloating problem, which is a known side effect of the Multi-Version Concurrency Control (MVCC) infrastructure. In our experiment, the size of the Accounts table increased by 0.6% in RC, 0.7% in SI, and 0.5% in SSI. In RC isolation, there are generally only two versions of data, whereas in SI, multiple versions may be retained depending on the duration of the transactions. This makes SI more susceptible to bloating. The extent of bloating becomes more significant in longer tests with both long and write-intensive transactions.

5.4. Experiment 4

In this experiment, our primary objective is to observe that the skewed write anomaly is detectable only under Serializable Snapshot Isolation (SSI), and our secondary goal is to assess its impact on system performance. For this purpose, we defined two transactions, labeled "tx3" and "tx4" (see Figure 11). Similar to the previous experiment, each customer in a single branch maintains two accounts—this time with 100 liras in the first account and 200 liras in the second account. Transactions tx3 and tx4 withdraw funds from the checking and investment accounts, respectively, ensuring that the total balance never falls below 0.

A constraint violation can occur because the constraint checked by one transaction can be altered by the other, and Snapshot Isolation (SI)

cannot detect this anomaly. The directed rows shown in Figure 11 represent the read-write conflicts (i.e. rw-antidependency), indicating that if the random aid value matches in each transaction, they cannot see each other's "abalance" updates on this aid account. To increase the likelihood of encountering skewed write error, transactions were executed with the same likelihood for the first 100 customers.

```
\set aid random(1, 100)
\set aid_offset (100000 * :scale /2)
\set delta random(1, 300)
BEGIN;

SELECT abalance AS b1 FROM accounts
WHERE aid=:aid \gset

SELECT abalance AS b2 FROM accounts
WHERE aid=:aid + :aid_offset \gset

\IF :b1 + :b2 - :delta > 0
        UPDATE accounts
        SET abalance = abalance - :delta
        WHERE aid = :aid;

\ENDIF
END;
```

```
\set aid random(100000 * :scale /2 +1,
                100000 * :scale /2 + 100)
\set aid_offset (100000 * :scale /2)
\set delta random(1, 300)
BEGIN;

SELECT abalance AS b2 FROM accounts
WHERE aid= :aid \gset

SELECT abalance AS b1 FROM accounts
WHERE aid=:aid - :aid_offset \gset

\IF :b1 + :b2 - :delta > 0
        UPDATE accounts
        SET abalance = abalance - :delta
        WHERE aid = :aid;

\ENDIF
END;
```

Figure 11. Transactions for Experiment 4

As shown in Figure 12, the average TPS rate under SI isolation is not worse than under RC, while the TPS of SSI is only impaired by 3% compared to RC. In RC, nearly 2 million transactions were completed without failures, and the balance became negative in only two accounts. In contrast, 5 out of nearly 2 million SI transactions failed, resulting in a negative balance in 3 accounts. The non-serializability observed in this case is not due to read-write conflicts, but rather to write-write conflicts between transactions of the same type.

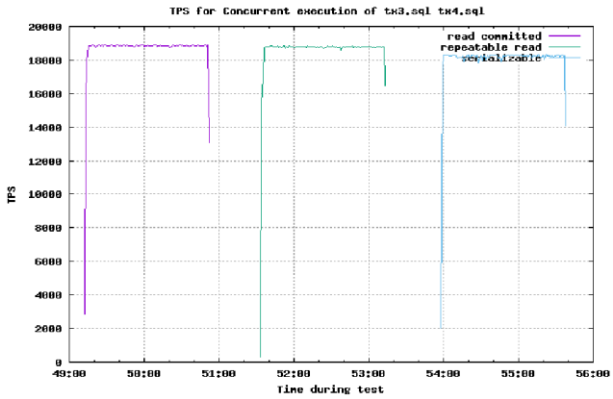


Figure 12. Outcomes of workloads containing skewed writes: system throughput

Under SSI isolation, 10 out of 1,819,632 transactions failed, but no account balances became negative. These failures were primarily due to write-write conflicts, while some were triggered by read-write conflicts that resulted in skewed writes.

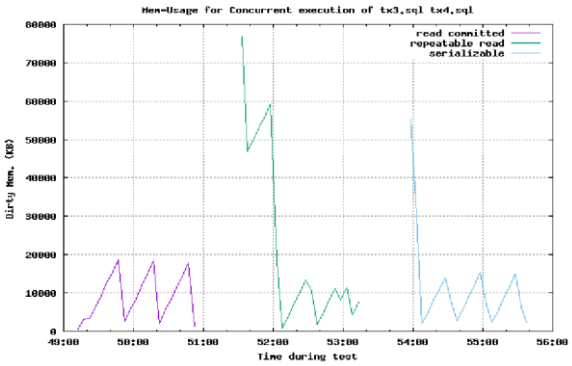


Figure 13. Outcomes of workloads containing skewed writes: memory consumption

We observed that memory usage remained consistent in RC and was more irregular in SI and SSI, though the differences were not substantial as seen in Figure 13. In workloads where the probability of skewed writes is higher, the increased memory consumption is likely due to the overhead of tracking these conflicts. Additionally, the bloating problem observed in this experiment was similar to that in the previous one.

5.5. Experiment 5

In this final experiment, our goal was to observe the behavior of different isolation levels across datasets of varying sizes. We executed two workloads—**tpc-b** and **no-branch**—with 8 clients on datasets of approximately 400 MB, 2 GB, 4 GB, 6 GB, and 8 GB, corresponding to scale factors of 20, 100, 200, 300, and 400, respectively. The experiment was repeated for each isolation level. Zendaoui & Hidouci (2015) emphasize similar points in their performance comparisons of SI and SSI under write-heavy workloads, specifically that the disk I/O metric is dominant in TPS. However, in this section, we conducted a more detailed analysis by evaluating some statistics related to buffer contention in PgSQL.

A common conclusion from both Figure 14 and 15 is that the impact of isolation levels on TPS diminishes as the dataset size increases. As data volume grows, memory space contention becomes a more dominant factor in determining TPS. This suggests an increased number of disk writes due to heightened buffer victim selection for larger datasets. To further investigate, Table 1 illustrates statistics related to buffer contention gathered during the **tpc-b** workload tests at the RC (test 4), Repeatable Read (RR) (test 9), and Serializable Snapshot Isolation (SSI) (test 14) isolation levels on a dataset scaled to 300 (approximately 6 GB).

Table 1. Measurements related to buffer contention

test	tps	ckpts	buf_clean (number of pages)	buf_backend (number of pages)	buf_alloc (number of pages)	wal_ written
4	3322	1	13,800	26,606	352,254	1,942,183,936
9	2927	1	11,200	30,373	321,384	1,717,403,648
14	3332	1	14,000	34,511	353,145	1,960,771,584

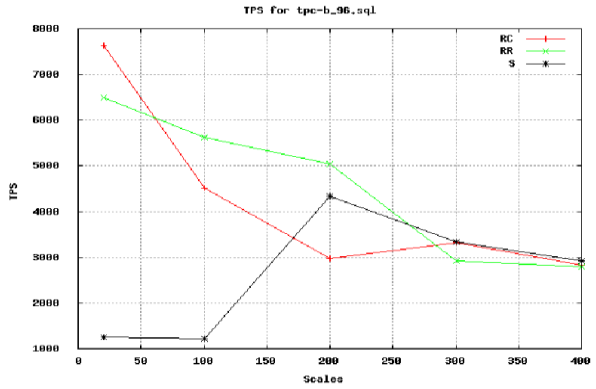


Figure 14. The effect of scalability on TPS for different isolation levels

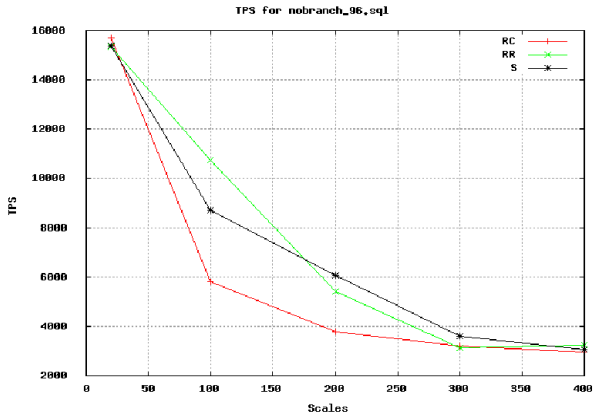


Figure 15. The effect of scalability on TPS for different isolation levels

In all three tests, checkpoints were observed. More importantly, the buf_alloc criterion indicates the amount of memory used during the test, and since this value exceeds the 2 GB memory limit, buffer victims were selected to free up space. Note that the page size in PgSQL is 8KB and 2 GB memory can hold 262,144 pages at most.

In PostgreSQL, the **bgwriter** (background writer) routine is responsible for managing the writing of dirty pages (modified data pages) from shared memory to disk. The buf_clean criterion in Table 1 reveals that the bgwriter routine selects victims due to memory bottlenecks, while the buf_backend metric shows that the client process had to allocate memory independently because the bgwriter's efforts were insufficient to resolve memory contention.

Another noteworthy point in the graphs is the difference in the behavior of SSI isolation between the **tpc-b** and **no-branch** workloads. The lower TPS rate of the **tpc-b** workload (under SSI isolation) for smaller datasets (400 MB and 2 GB) is attributed to the increased number of failed transactions caused by write-write contention in small tables such as Branches and Tellers.

For instance, in both tests, approximately 25% of transactions failed and had to be retried an average of nearly 20 times. However, as the dataset size increases, this contention decreases, and performance under SSI isolation becomes similar to that of other isolation levels.

As previously mentioned, memory contention becomes the dominant factor affecting performance as the data volume grows. Since the no-branch workload does not experience the same level of intense conflicts, its performance under SSI isolation remains comparable to other isolation levels across all dataset scales.

6. DISCUSSION AND CONCLUSION

This study provides a general overview of database concurrency control methods. We begin with a conceptual examination of current solutions, focusing on serializability, in-memory serial execution, weak isolation levels, and the increasingly important Snapshot Isolation (SI) and Serializable Snapshot Isolation (SSI) levels, including their advantages and disadvantages. Finally, we analyze the implementation principles of the MVCC model and isolation levels within PostgreSQL.

Our empirical study evaluates the accuracy, throughput, and resource utilization of different isolation levels. Building a robust experimental framework for such comparisons is challenging; however, PostgreSQL's *pgbench* application provides a convenient and realistic solution. Our experiments first investigated optimal system configuration settings for various workloads with differing anomaly characteristics (e.g., skewed read/write ratios). The experimental results of this study can be summarized as follows:

- The RC method performs significantly better in heavily updated OLTP-style workloads compared to SI and SSI due to their high rollback rates. (Experiment 2).
- The advantages of snapshot methods in mixed workloads containing both OLTP

and OLAP movements have been confirmed (Experiment 3).

- The SSI method has been validated to detect write skew anomalies without compromising performance (Experiment 4).
- The impact of isolation level on TPS diminishes with increasing data volume, with memory contention becoming the dominant factor affecting TPS. This suggests that the performance overhead of SSI becomes negligible compared to weaker isolation levels for large datasets (Experiment 5).

As mentioned at the beginning of Section 3, there are significant expectations regarding the scalability of SI and SSI methods in the near future. Many studies shed light on this perspective. For instance, Binnig et al. (2014) emphasize at the end of their work that the incremental approach they developed for SI in distributed datasets can also be adapted to SSI. Another example is PolarDB-X, which is currently used as a database service at Alibaba Cloud (Cao et al., 2022). The study discusses ways to implement the SI method in distributed databases at a lower coordination cost.

We believe this paper combines theoretical analysis with practical results, offering a valuable resource for database administrators seeking to improve their understanding and practice. Future studies will extend this work by conducting similar experiments on different database systems using benchmark environments such as HammerDB (URL-5), YCSB(URL-6), and similar platforms. We also plan to investigate and test current concurrency protocols in distributed databases, particularly using YugabyteDB, a fully open-source PostgreSQL-compatible system, to explore the behavior of these protocols on larger datasets.

REFERENCES

Adya, A., Liskov, B.H., & O'Neil, P.E. (2000). Generalized isolation level definitions. Proceedings of 16th International Conference on Data Engineering (Cat. No.00CB37073), 67-78.

Almeida, D., Lopes, M., Saraiva, L., Abbasi, M., Martins, P., Silva, J.C., & Váz, P. (2023). Performance Comparison of Redis, Memcached, MySQL, and PostgreSQL: A Study on Key-Value and Relational Databases. 2023 Second International Conference On Smart Technologies For Smart Nation (SmartTechCon), 902-907.

Alomari, M.A., Fekete, A.D., & Röhm, U. (2014). Performance of program modification techniques that ensure serializable executions with snapshot isolation DBMS. *Inf. Syst., 40*, 84-101.

ANSI. Database language – SQL. American National Standard for Information Systems X3.135-1992, (1992).

Beillahi, S.M., Bouajjani, A., & Enea, C. (2021). Checking Robustness Between Weak Transactional Consistency Models. Programming Languages and Systems, 12648, 87 - 117.

Berenson, H. & Bernstein, P. & Gray, J. & Melton, J. & O'Neil, E. & O'Neil, P. (1995). A Critique Of ANSI SQL İsolation Levels. pp. 1–10, arXiv:cs/0701157,doi:10.1145/223784.223785, ISBN 978-0897917315. ACM SIGMOD international Conference on Management of Data

Bernstein, P. A. & Goodman, N. (1981). Concurrency Control İn Distributed Database Systems, ACM Computing Surveys

Binnig, C., Hildenbrand, S., Färber, F., Kossmann, D., Lee, J., & May, N. (2014). Distributed snapshot isolation: global transactions pay globally, local transactions pay locally. The VLDB Journal, 23, 987 - 1011.

Bog, A., Kruger, J., & Schaffner, J. (2008). A Composite Benchmark for Online Transaction Processing and Operational Reporting. 2008 IEEE Symposium on Advanced Management of Information for Globalized Enterprises (AMIGE), 1-5.

Cahill, M.J. & Röhm, U. & Fekete, A. (2008). Serializable isolation for snapshot databases. ACM(SIGMOD),doi:10.1145/1376616.1376690

Cao, W., Li, F., Huang, G., Lou, J., Zhao, J., He, D., Sun, M., Zhang, Y., Wang, S., Wu, X., Liao, H., Chen, Z., Fang, X., Chen, M., Liang, C., Luo, Y., Wang, H., Wang, S., Ma, Z., Yang, X., Peng, X., Ruan, Y., Wang, Y., Zhou, J., Wang, J., Hu, Q., & Kang, J. (2022). PolarDB-X: An Elastic Distributed Relational Database for Cloud-Native Applications. 2022 IEEE 38th International Conference on Data Engineering (ICDE), 2859-2872.

Fekete, A.D., O'Neil, E.J., & O'Neil, P.E. (2004). A read-only transaction anomaly under snapshot isolation. SIGMOD Rec., 33, 12-14.

Filip, P., & Cegan, L. (2020). Comparison of MySQL and MongoDB with focus on performance. 2020 International Conference on Informatics, Multimedia, Cyber and Information System (ICIMCIS), 184-187.

Kingsbury, K., & Alvaro, P. (2020). Elle: Inferring Isolation Anomalies from Experimental Observations. ArXiv, abs/2003.10554.

Kleppmann, M. (2017). Designing Data-Intensive Application, Publisher(s): O'Reilly Media, Inc. ISBN: 9781491903100

Li, K., Weng, S., Ni, L., Yang, C., Zhang, R., Zhou, X., & Zhou, A. (2024). DBStorm: Generating Various Effective Workloads for Testing Isolation Levels. International Symposium on Software Testing and Analysis.

Liarokapis, D., O'Neil, E.J., & O'Neil, P.E. (2020). Testing Concurrency in Databases still Matters. 2020 International Conference on Information Technologies (InfoTech), 1-6.

Ports, D. R. K., Grittner, K., (2012). Serializable snapshot isolation in PostgreSQL. VLDB, https://doi.org/10.48550/arXiv.1208.4179. Proceedings of the VLDB Endowment (PVLDB), Vol. 5, No. 12, pp. 1850-1861

Salunke, S.V., & Ouda, A. (2024). A Performance Benchmark for the PostgreSQL and MySQL Databases. Future Internet.

Silberschatz, A. & Korth, H. F. & Sudarshan, S. (2019) Database System Concepts, Seventh Edition, McGraw-Hill ISBN 9780078022159

URL-1:http://www.bailis.org/blog/when-is-acid-acid-rarely/#acidtable [Accessed June, 2024]

URL-2:http://www.interdb.jp/pg/pgsql05.html [Accessed June, 2024]

URL-3:https://postgrespro.com/blog/Rogov [Accessed June, 2024]

URL-4:Yahoo!CloudServingBenchmark(YCSB) https://github.com/brianfrankcooper/YCSB/ [Accessed June, 2024]

URL-5:HammerDB, https://www.hammerdb.com/download.html [Accessed June, 2024]

URL-6:pgbench, https://www.postgresql.org/docs/current/pgbench.html [Accessed June, 2024]

URL-7:TPC-BHomepage, https://www.tpc.org/tpcb/ [Accessed June, 2024]

URL-8:TuningYourPostgreSQLServer, https://wiki.postgresql.org/wiki/Tuning_Your_PostgreSQL_Server [Accessed June, 2024]

URL-9:PostgreSQL Benchmarking Toolset, https://github.com/gregs1104/pgbench-tools [Accessed Sept. 28, 2023]

Wang, T., Johnson, R., Fekete, A.D., & Pandis, I. (2016). Efficiently making (almost) any concurrency control mechanism serializable. *The VLDB Journal, 26*, 537 - 562.

Wiechork, K., & Charão, A.S. (2020). Investigating the Performance of Moodle Database Queries in Cloud Environments. International Conference on Enterprise Information Systems.

Tongkaw, S., & Tongkaw, A. (2016). A comparison of database performance of MariaDB and MySQL with OLTP workload. 2016 IEEE Conference on Open Systems (ICOS), 117-119.

Zendaoui, F., & Hidouci, W. (2015). Performance evaluation of serializable snapshot isolation in PostgreSQL. 2015 12th International Symposium on Programming and Systems (ISPS), 1-11.