



Department of Information and Electronic Engineering

Final year thesis

MVCC in the Modern RDBMS

Student
KONSTANTINOS DIAMANTIDIS
Student ID: it083389

Supervisor
PROFESSOR DIMITRIS A. DERVOS

June 2020

Τίτλος Π.Ε.: Η Τεχνολογία MVCC στα Σύγχρονα Περιβάλλοντα RDBMS

Κωδικός Π.Ε.: 20156

Ονοματεπώνυμο φοιτητή: Κωνσταντίνος Διαμαντίδης

Ονοματεπώνυμο επιβλέποντα: Δημήτρης Α. Δέρβος

Ημερομηνία ανάληψης Π.Ε.: 9 Απριλίου 2020

Ημερομηνία περάτωσης Π.Ε.: 16 Ιουνίου 2020

Βεβαιώνω ότι είμαι ο συγγραφέας αυτής της εργασίας και ότι κάθε βοήθεια την οποία είχα για την προετοιμασία της είναι πλήρως αναγνωρισμένη και αναφέρεται στην εργασία. Επίσης, έχω καταγράψει τις όποιες πηγές από τις οποίες έκανα χρήση δεδομένων, ιδεών, εικόνων και κειμένου, είτε αυτές αναφέρονται ακριβώς είτε παραφρασμένες. Επιπλέον, βεβαιώνω ότι αυτή η εργασία προετοιμάστηκε από εμένα προσωπικά, ειδικά ως πτυχιακή εργασία, στο Τμήμα Μηχανικών Πληροφορικής και Ηλεκτρονικών Συστημάτων του ΔΙ.ΠΑ.Ε.

Η παρούσα εργασία αποτελεί πνευματική ιδιοκτησία του φοιτητή Κωνσταντίνου Διαμαντίδη που την εκπόνησε. Στο πλαίσιο της πολιτικής ανοικτής πρόσβασης, ο συγγραφέας/δημιουργός εκχωρεί στο Διεθνές Πανεπιστήμιο της Ελλάδος άδεια χρήσης του δικαιώματος αναπαραγωγής, δανεισμού, παρουσίασης στο κοινό και ψηφιακής διάχυσης της εργασίας διεθνώς, σε ηλεκτρονική μορφή και σε οποιοδήποτε μέσο, για διδακτικούς και ερευνητικούς σκοπούς, άνευ ανταλλάγματος. Η ανοικτή πρόσβαση στο πλήρες κείμενο της εργασίας, δεν σημαίνει καθ' οιονδήποτε τρόπο παραχώρηση δικαιωμάτων διανοητικής ιδιοκτησίας του συγγραφέα/δημιουργού, ούτε επιτρέπει την αναπαραγωγή, αναδημοσίευση, αντιγραφή, πώληση, εμπορική χρήση, διανομή, έκδοση, μεταφόρτωση (downloading), ανάρτηση (uploading), μετάφραση, τροποποίηση με οποιονδήποτε τρόπο, τμηματικά ή περιληπτικά της εργασίας, χωρίς τη ρητή προηγούμενη έγγραφη συναίνεση του συγγραφέα/δημιουργού.

Η έγκριση της πτυχιακής εργασίας από το Τμήμα Μηχανικών Πληροφορικής και Ηλεκτρονικών Συστημάτων του Διεθνούς Πανεπιστημίου της Ελλάδος, δεν υποδηλώνει απαραίτητα και αποδοχή των απόψεων του συγγραφέα, εκ μέρους του Τμήματος.

Thesis title: MVCC in the Modern RDBMS
Thesis code number: 20156
Student full name: Konstantinos Diamantidis
Supervisor full name: Dimitris A. Dervos
Thesis assignment date: 9th April 2020
Thesis completion date: 16th June 2020

I hereby declare that I am the original author of this thesis and that every assistance that I received in the course of the project work reported is acknowledged and (where possible) cited in the "References/Bibliography" section of the present manuscript. All sources of information and/or data, ideas, pictures, graphs, figures and text excerpts used (be it "as is" quotations, or re-phrased), they are properly referenced in footnotes or directly in text. I also declare that, I myself have conducted the project work herewith reported and compiled the present manuscript, along the lines of fulfilling the BSc thesis project requirement of my study with the Department of Information and Electronic Systems Engineering.

The present BSc thesis manuscript and the project work reported comprise the intellectual property of the author, Mr. Konstantinos Diamantidis. In the context of an Open Access Mandate¹, the author grants the International Hellenic University the permission to reproduce, lend, display publicly, and make it freely available for educational and/or research purposes. Open access to the full text of the thesis, does not imply, in any way, permission to abandon the author's intellectual property rights on the work reported, nor does it imply permission to reproduce, copy, republish, generate profit from, commercially exploit, re-distribute via platforms not belonging to the International Hellenic University, translate, edit small or extended parts of its content in any form, without the explicit permission from the author and his written consent.

The approval of this thesis manuscript and the project work reported should not be taken to imply that the Department of Information and Electronic Engineering, the School of Engineering, and the International Hellenic University share the personal views expressed by the author, nor should it be taken to imply that they are liable for his reported scientific and/or experimental findings.

¹https://en.wikipedia.org/wiki/Open-access_mandate

Περίληψη

Η παρούσα πτυχιακή εργασία παρουσιάζει τους μηχανισμούς που υλοποιούν την τεχνολογία Multi-Version Concurrency Control (MVCC) σε τέσσερα από τα μεγαλύτερα Σχεσιακά Συστήματα Διαχείρισης Βάσεων Δεδομένων (ΣΣΔΒΔ), αυτά των Oracle, MySQL, PostgreSQL και SQL Server. Ένα κεφάλαιο αφιερώνεται σε κάθε ΣΣΔΒΔ, όπου αναλύεται η προσέγγισή του σχετικά με την υλοποίηση της τεχνολογίας MVCC. Επίσης, τα επιπέδα απομόνωσης συναλλαγών, που υλοποιούνται με την τεχνολογία MVCC, ελέγχονται για έξι από τα γνωστά φαινόμενα που συναντώνται κατά την προσπάθεια απομόνωσης και του συγχρονισμού των συναλλαγών στα ΣΣΔΒΔ. Προς διευκόλυνση της διαδικασίας αυτού του ελέγχου, αναπτύχθηκε το λογισμικό MVCC_sim.py χρησιμοποιώντας την γλώσσα προγραμματισμού Python που συμπεριλαμβάνεται στα παραδοτέα της παρούσας εργασίας. Ο πρωταρχικός στόχος της όλης προσπάθειας που έγινε δεν αφορά μόνον στην μελέτη των τρόπων υλοποίησης της τεχνολογίας MVCC από τα τέσσερα δημοφιλέστερα ΣΣΔΒΔ, αλλά και στη σύνταξη μιας συνοπτικής εικόνας των κοινών και των μή κοινών χαρακτηριστικών των τελευταίων στην εφαρμογή της συγκεκριμένης τεχνολογίας. Τα ευρήματα αυτής της προσπάθειας συνοψίζονται στον πίνακα 6.1 του 6ου κεφαλαίου.

Abstract

The current thesis presents the underlying mechanisms that implement Multi-Version Concurrency Control in four major Relational Database Management Systems, those of Oracle, MySQL, PostgreSQL and Microsoft's SQL Server. A separate chapter is dedicated to each one of the RDBMSs considered. In the beginning of each chapter, the MVCC technology stack is described. Next, the RDBMS's isolation levels which use the MVCC technology are considered for each one of the six known concurrency control anomalies. The MVCC_sim.py application developed along the lines of this project has been developed in the Python programming language. The primary objective of the projet has been to not only consider MVCC and the way it is implemented in the most popular RDBMS environments, but to provide a concise view on what the latter have in common, and in what they differ when it comes to implementing concurrency control for DBMS transactions. Table 6.1 in chapter 6 summarises the findings of the "learn and prove by verifying" approach adopted and used in the project.

Acknowledgements

I wish to express my sincere gratitude to my supervisor, Professor Dimitris A. Dervos, for entrusting me with this BSc thesis theme, and for his invaluable guidance and constructive interaction throughout the project. My special gratitude I wish to extend to include Mr. Martti Laiho of Haaga-Helia University in Helsinki, Finland. The shaping of the present treatise and the overall "learn and prove by verifying" approach reported herewith are heavily due to Mr. Laiho's work and writings. I also wish to thank all those who devoted time to read this manuscript, as it has been evolving in the form of subsequent versions, and provided invaluable feedback.

Contents

List of Tables	i
List of Figures	iv
1 Introduction	1
1.1 DBMS transaction	2
1.2 The need for concurrency and concurrency control	2
1.3 Concurrency Control (CC) protocols	3
1.4 MVCC	3
1.5 Scope of current thesis	4
1.6 Contribution of current thesis	5
2 MVCC in Oracle	7
2.1 Oracle Transaction Isolation Levels	8
2.1.1 READ COMMITTED	8
2.1.2 SERIALIZABLE	8
2.1.3 READ ONLY	9
2.2 Oracle and MVCC	9
2.2.1 Transaction lifecycle	9
2.3 Examples of isolation levels	12
2.3.1 Non-repeatable read and READ COMMITTED	12
2.3.2 Non-repeatable read and SERIALIZABLE	13
2.3.3 Lost update and READ COMMITTED	14
2.3.4 Lost update and SERIALIZABLE	15
2.3.5 Ghost update and READ COMMITTED	16
2.3.6 Ghost update and SERIALIZABLE	17
2.3.7 Phantom update and READ COMMITTED	18
2.3.8 Phantom update and SERIALIZABLE	19
2.3.9 Read skew and READ COMMITTED	20
2.3.10 Read skew and SERIALIZABLE	21
2.3.11 Write skew and SERIALIZABLE	22
2.3.12 Write skew and READ COMMITTED	24
2.4 Comparison of test results	25

3	MVCC in MySQL	27
3.1	MySQL Transaction Isolation Levels	28
3.1.1	READ COMMITTED	28
3.1.2	REPEATABLE READ	29
3.2	MySQL and MVCC	30
3.3	Examples of isolation levels	32
3.3.1	Non-repeatable read and READ COMMITTED	32
3.3.2	Non-repeatable read and REPEATABLE READ	33
3.3.3	Lost update and READ COMMITTED	34
3.3.4	Lost update and REPEATABLE READ	35
3.3.5	Ghost update and READ COMMITTED	36
3.3.6	Ghost update and REPEATABLE READ	37
3.3.7	Phantom update and READ COMMITTED	38
3.3.8	Phantom update and REPEATABLE READ	39
3.3.9	Read skew and READ COMMITTED	40
3.3.10	Read skew and REPEATABLE READ	41
3.3.11	Write skew and REPEATABLE READ	42
3.3.12	Write skew and READ COMMITTED	44
3.4	Comparison of test results	45
4	MVCC in PostgreSQL	47
4.1	PostgreSQL Transaction Isolation Levels	48
4.1.1	READ COMMITTED	48
4.1.2	REPEATABLE READ	49
4.1.3	SERIALIZABLE	49
4.2	PostgreSQL and MVCC	49
4.3	Examples of isolation levels	51
4.3.1	Non-repeatable read and READ COMMITTED	51
4.3.2	Non-repeatable read and REPEATABLE READ	52
4.3.3	Non-repeatable read and SERIALIZABLE	52
4.3.4	Lost update and READ COMMITTED	53
4.3.5	Lost update and REPEATABLE READ	54
4.3.6	Lost update and SERIALIZABLE	55
4.3.7	Ghost update and READ COMMITTED	56
4.3.8	Ghost update and REPEATABLE READ	57
4.3.9	Ghost update and SERIALIZABLE	58
4.3.10	Phantom update and READ COMMITTED	59
4.3.11	Phantom update and REPEATABLE READ	60
4.3.12	Phantom update and SERIALIZABLE	61
4.3.13	Read skew and READ COMMITTED	62
4.3.14	Read skew and REPEATABLE READ	63
4.3.15	Read skew and SERIALIZABLE	64
4.3.16	Write skew and REPEATABLE READ	65
4.3.17	Write skew and SERIALIZABLE	67
4.3.18	Write skew and READ COMMITTED	68

4.4	Comparison of test results	69
5	MVCC in SQL Server	71
5.1	SQL Server Transaction Isolation Levels	72
5.1.1	READ COMMITTED with Snapshot Isolation	72
5.1.2	SNAPSHOT	72
5.2	SQL Server and MVCC	73
5.3	Examples of isolation levels	74
5.3.1	Non-repeatable read and READ COMMITTED	74
5.3.2	Non-repeatable read and SNAPSHOT	75
5.3.3	Lost update and READ COMMITTED	76
5.3.4	Lost update and SNAPSHOT	77
5.3.5	Ghost update and READ COMMITTED	78
5.3.6	Ghost update and SNAPSHOT	79
5.3.7	Phantom update and READ COMMITTED	80
5.3.8	Phantom update and SNAPSHOT	81
5.3.9	Read skew and READ COMMITTED	82
5.3.10	Read skew and SNAPSHOT	83
5.3.11	Write skew and SNAPSHOT	84
5.3.12	Write skew and READ COMMITTED	86
5.4	Comparison of test results	87
6	Conclusion	89
	References	92
	Appendix A MVCC_sim User Guide	93

List of Tables

2.1	Oracle isolation levels and transaction anomalies	25
3.1	MySQL isolation levels and transaction anomalies	45
4.1	PostgreSQL isolation levels and transaction anomalies	69
5.1	SQL Server isolation levels and transaction anomalies	87
6.1	Transaction anomalies in all four RDBMSs	90

List of Figures

2.1	Read consistency in the READ COMMITTED isolation level[1]	10
2.2	Non-repeatable read in Oracle's READ COMMITTED isolation level	12
2.3	Non-repeatable read in Oracle's SERIALIZABLE isolation level	13
2.4	Lost update in Oracle's READ COMMITTED isolation level	14
2.5	Lost update in Oracle's SERIALIZABLE isolation level	15
2.6	Ghost update in Oracle's READ COMMITTED isolation level	16
2.7	Ghost update in Oracle's SERIALIZABLE isolation level	17
2.8	Phantom update in Oracle's READ COMMITTED isolation level	18
2.9	Phantom update in Oracle's SERIALIZABLE isolation level	19
2.10	Read skew in Oracle's READ COMMITTED isolation level	20
2.11	Read skew in Oracle's SERIALIZABLE isolation level	21
2.12	Example of write skew occurring (all false) [2]	22
2.13	Write skew in Oracle's SERIALIZABLE isolation level	23
2.14	Write skew in Oracle's READ COMMITTED isolation level	24
3.1	MySQL Undo Log chaining representation [3]	30
3.2	Non-repeatable read in MySQL's READ COMMITTED isolation level	32
3.3	Non-repeatable read in MySQL's REPEATABLE READ isolation level	33
3.4	Lost update in MySQL's READ COMMITTED isolation level	34
3.5	Lost update in MySQL's REPEATABLE READ isolation level	35
3.6	Ghost update in MySQL's READ COMMITTED isolation level	36
3.7	Ghost update in MySQL's REPEATABLE READ isolation level	37
3.8	Phantom update in MySQL's READ COMMITTED isolation level	38
3.9	Phantom update in MySQL's REPEATABLE READ isolation level	39
3.10	Read skew in MySQL's READ COMMITTED isolation level	40
3.11	Read skew in MySQL's REPEATABLE READ isolation level	41
3.12	Example of write skew occurring (all false) [2]	42
3.13	Write skew in MySQL's REPEATABLE READ isolation level	43
3.14	Write skew in MySQL's READ COMMITTED isolation level	44
4.1	PostgreSQL's UPDATE under the hood [4]	50
4.2	Non-repeatable read in Postgres's READ COMMITTED isolation level	51
4.3	Non-repeatable read in Postgres's REPEATABLE READ isolation level	52
4.4	Non-repeatable read in Postgres's SERIALIZABLE isolation level	52
4.5	Lost update in Postgres's READ COMMITTED isolation level	53

4.6	Lost update in Postgres's REPEATABLE READ isolation level	54
4.7	Lost update in Postgres's SERIALIZABLE isolation level	55
4.8	Ghost update in Postgres's READ COMMITTED isolation level	56
4.9	Ghost update in Postgres's REPEATABLE READ isolation level	57
4.10	Ghost update in Postgres's SERIALIZABLE isolation level	58
4.11	Phantom update in Postgres's READ COMMITTED isolation level	59
4.12	Phantom update in Postgres's READ COMMITTED isolation level	60
4.13	Phantom update in Postgres's SERIALIZABLE isolation level	61
4.14	Read skew in Postgres's READ COMMITTED isolation level	62
4.15	Read skew in Postgres's REPEATABLE READ isolation level	63
4.16	Read skew in Postgres's SERIALIZABLE isolation level	64
4.17	Example of Write Skew occurring (all false) [2]	65
4.18	Write skew in Postgres's REPEATABLE READ isolation level	66
4.19	Write skew in Postgres's SERIALIZABLE isolation level	67
4.20	Write skew in Postgres's READ COMMITTED isolation level	68
5.1	Non-repeatable read in SQL Server's READ COMMITTED isolation level . . .	74
5.2	Non-repeatable read in SQL Server's SNAPSHOT isolation level	75
5.3	Lost update in SQL Server's READ COMMITTED isolation level	76
5.4	Lost update in SQL Server's SNAPSHOT isolation level	77
5.5	Ghost update in SQL Server's READ COMMITTED isolation level	78
5.6	Ghost update in SQL Server's SNAPSHOT isolation level	79
5.7	Phantom update in SQL Server's READ COMMITTED isolation level	80
5.8	Phantom update in SQL Server's SNAPSHOT isolation level	81
5.9	Read skew in SQL Server's READ COMMITTED isolation level	82
5.10	Read skew in SQL Server's SNAPSHOT isolation level	83
5.11	Example of write skew occurring (all false) [2]	84
5.12	Write skew in SQL Server's SNAPSHOT isolation level	85
5.13	Write skew in SQL Server's READ COMMITTED isolation level	86
A.1	Initial screen where user can select their preferred DBMS	94
A.2	Test scenario selection	95
A.3	Test scenario executed	96
A.4	YAML file with the test#2 steps	97

Chapter 1

Introduction

1.1 DBMS transaction

The origin of transactions comes from contract law. Legal transactions must be:

- **atomic**, either all provisions apply or none at all
- **consistent**, all transactions and provisions apply by the same legal protocols
- **durable**, when parties commit to the transaction, they cannot back off

These properties are the "A", "C" and "D" in the popular "ACID" acronym for Database Management Systems (DBMSs), the "I" standing for **isolation**.

In databases, as opposed to law, a transaction is composed of operations that transform the database from one **consistent** state to another. This means that if all database consistency constraints were satisfied prior to running a transaction, then they will remain satisfied afterward [5].

A transaction executes in an "all or nothing at all" fashion (**atomicity**) and, when successful, modified data persist to non-volatile memory so as to be available even in case of a power failure (**durability**).

The concept of **isolation** is introduced to ensure that when many transactions are running simultaneously, the end result would be equivalent to that if they had run serially. This thesis explores how each DBMS ensures this property.

1.2 The need for concurrency and concurrency control

One could imagine database transactions running serially, each waiting its turn to exclusively access the data it needs. The database would smoothly move from one consistent state to another without worrying about conflicts. However, the utopia of serialized transactions is infeasible for virtually any high-performance, multi-user system [5].

With the advent of CPU and I/O parallelisation multiple transactions can be processed concurrently. Idle time of disks and processors is decreased and the average number of transactions completed in a given time is increased. So, for example, interleaving the execution of a short-running transaction with a long-running one, will usually allow the short transaction to complete faster than if they had run serially.

However, running transactions concurrently brings along dangers. Without proper oversight, transactions can interfere with each other's data. In such scenarios, incorrect query results, constraints violations and general data integrity problems arise.

Take a regular wire transfer for example. If one transaction reads the bank account balance when the money have been withdrawn from the original account and before they have been deposited to the destination account, it would seem as the money disappeared from the bank. The property of **isolation** would have been breached. Similar examples apply in many more situations, for example in all sorts of orders, payments or reservation/ticketing systems.

1.3 Concurrency Control (CC) protocols

In Database Management Systems, isolation is guaranteed by using a concurrency control protocol. Traditionally, the problem of concurrency is solved using locking. If transaction A needs access to resource N, it locks it; after use the lock is released. If transaction B wants to access resource N while A is using it, it must wait [6].

The Multiple Granularity Locking (MGL) scheme was "presented and implemented in the early relational prototype database System R1, and has since been copied and modified as the de facto locking scheme in later DBMS products, such as Db2, Oracle, and the DBMS products derived from the Esvel 2 engine, for example HP Allbase, Tandem NonStopSQL, Sybase and Microsoft SQL Server" [7]. The two-phase locking (2PL) method [8] is used in order to guarantee the serializability of the transactions. Each transaction acquires the proper locks on the tuples it needs before it is allowed to read or modify them.

Modern database concurrency control has since progressed and protocols like strict 2-phase locking can guarantee concurrency and strong serializability. However, any system that depends on this sort of concurrency control suffers an inherent scalability problem. By design, read/write locks prevent readers and writers from running simultaneously. So, as the volume of reads and writes increase, one runs into situations where there is unnecessary queueing, or either the readers or writers get starved¹ [9].

1.4 MVCC

The first mention of Multi-Version Concurrency Control (MVCC) appeared in a 1979 PhD thesis [10] and the first implementation started in 1981 [11] for the InterBase DBMS (now open-sourced as Firebird).

The primary concept of MVCC is that the DBMS maintains multiple physical versions of each logical object in the database, in order to allow operations on the same object to proceed in parallel. Because of this, each user connected to the database sees a snapshot of the database at a particular instant in time. Any changes made by a writer will not be seen by other users until the running transaction commits successfully. The main advantage of using the MVCC model of concurrency control rather than pure locking (e.g., MGL) is that in MVCC, *locks acquired for reading data do not conflict with locks acquired for writing data, and so reading never blocks writing and writing never blocks reading* [12].

MVCC is probably the most popular concurrency control protocol among modern RDBMSs. Although it was discovered in the late 1970s, it is still used in almost every major RDBMS. Despite its popularity, there is no one "standard" implementation. Instead, there are several design choices that have different trade-offs and performance behaviors [13].

¹[https://en.wikipedia.org/wiki/Starvation_\(computer_science\)](https://en.wikipedia.org/wiki/Starvation_(computer_science))

1.5 Scope of current thesis

The intention of this thesis is to explore the underlying MVCC implementation in four of the major modern RDBMSs. The most popular RDBMSs are considered, namely Oracle, MySQL, PostgreSQL and Microsoft's SQL Server.

Each RDBMS's MVCC approach is presented, and, in addition, the isolation levels implemented via MVCC are tested against the following six transaction anomalies:

- **Non-repeatable read**, also known as *Fuzzy read*, occurs when a value that has been read by a (still active) transaction is overwritten by a concurrent transaction and, as a result, a subsequent read would see a different value than the one it read earlier in the transaction.
- **Lost update**, occurs when the updated data of a transaction get overwritten by the update operations of another concurrent transaction, disregarding the former transaction's update.
- **Ghost update**, occurs when a transaction tries to update a row that got deleted by a concurrent transaction and no warning or exception is thrown.
- **Phantom update**, occurs when a transaction successfully updates data that did not exist when it began, but got inserted in the mean time by a concurrent transaction.
- **Read skew**, happens because of the non-repeatable read functionality of certain isolation levels. In essence, a transaction will read partial data before a concurrent transaction modifies them and the rest after the modification takes place. This can result in data integrity constraints violations. For example, a transaction reads the value of two bank accounts after another concurrent transaction withdrew an amount but before it deposited it.
- **Write skew**, happens because of the repeatable read functionality of certain isolation levels. In essence, a transaction, when evaluating a condition before writing data, will not take into account a necessary change by a concurrent transaction. Thus, the condition will falsely evaluate and the former transaction will proceed with writing the data (skewed).

1.6 Contribution of current thesis

The thesis is based on and builds upon the foundation knowledge and practice laid by in [7]. In addition to the latter, a wide range of literature is considered on all four RDBMSs, that includes, but is not limited to, all four RDBMSs (Oracle, MySQL, PostgreSQL, SQL Server). The relevant information was organised in a way that aims to clarify the MVCC technology stack and how the isolation levels behave when encountering transaction anomalies for each RDBMS. At the end of each chapter there is a table that sums up the behaviour of each MVCC-based isolation level and in the last chapter there is a summarizing table (Table 6.1) that unifies all the previous tables, so that one can compare the results across the RDBMSs used.

Last but not least, a Python application has been created in order to automate the testing of all the relevant SQL transaction examples involving up to three concurrent RDBMS sessions. The application accepts SQL queries as text input and executes them in a pre-specified order across the concurrent sessions presented side-by-side to the user. The application supports all four RDBMSs used and demonstrates the way they implement transaction isolation levels and the way they handle transaction concurrency anomalies. The figures in all examples in chapters 2-5 comprise snapshots taken when running the `MVCC_sim.py` application. Appendix A comprises a brief user guide for the `MVCC_sim.py` application.

Chapter 2

MVCC in Oracle

2.1 Oracle Transaction Isolation Levels

While ANSI SQL 92 defines four levels of isolation, Oracle implements only two of them, READ COMMITTED, which is the default option, and SERIALIZABLE. Oracle, also implements READ ONLY isolation level which is not defined in the standard and will not be examined in this thesis. All three transaction isolation levels (READ COMMITTED, SERIALIZABLE and READ ONLY) are briefly outlined in the following sections.

2.1.1 READ COMMITTED

In the READ COMMITTED isolation level, every query executed by a transaction sees only data committed before the query (not the transaction) began. Every query is provided with transaction set consistency, meaning that queries see data in a consistent state[14].

The READ COMMITTED isolation level is more widely used and suitable for environments that don't have the need to issue the same query twice and have low transaction arrival rate, which results in lower risk of incorrect results due to lost updates, phantoms or non-repeatable reads.

Other use cases for this isolation level, are environments that require the faster response times that READ COMMITTED achieves when compared to the SERIALIZABLE isolation level.

2.1.2 SERIALIZABLE

In the SERIALIZABLE isolation level, a transaction sees only changes committed at the time the transaction (not the query) began and changes made by the transaction itself. Any row read by the transaction is guaranteed to remain the same when it is reread, provided that the transaction in question has not altered its data content in the meantime. Any query is guaranteed to return the same results for the duration of the transaction, so changes made by other transactions are not visible to the query regardless of how long it has been running[14].

The SERIALIZABLE isolation level is most suitable for situations where the chance of two concurrent transactions modifying the same rows is low. Preferably used in situations where the most concurrently running transactions are either read-only or update a small number of rows.

An application that would use the SERIALIZABLE isolation level could be one of a report-writing type that generates summary data and stores it in the database. Application developers, must put extra effort when using this isolation level, as they need to check for the "Cannot serialize access" error and handle the transaction properly (e.g., rollback or retry).

An example of an environment that should avoid implementing the SERIALIZABLE isolation level in its transactions, is one of long running transactions that must update the same rows, accessed by a high volume of short update transactions. Time and effort will be wasted by repeated rollbacks due to the fact that long running transactions most likely would not be the first to modify a given row.

The SERIALIZABLE isolation level provides more consistency by offering protection from anomalies like the lost updates, phantom updates or non-repeatable reads.

2.1.3 READ ONLY

READ ONLY is equivalent to SERIALIZABLE, except for the fact that INSERT, UPDATE and DELETE operations are not allowed. This isolation level is used primarily for reporting/analytics purposes.

2.2 Oracle and MVCC

Oracle RDBMS always enforces **statement-level read consistency**[14]. This means that a query is guaranteed to return data from a specific point in time, this is the time that the query took place. That way, a query never reads dirty data or changes made by transactions that committed while the query was being processed. As query execution proceeds, only data committed before the query began is visible to the it.

There is also the optional choice of enforcing **transaction-level read consistency**[14]. When a transaction runs in SERIALIZABLE mode, all data accesses reflect the state of the database as of the time that the transaction began and not the time each subsequent query got processed. Thus, the data seen by all queries within the same transaction is consistent with respect to a single point in time, this is the time that the transaction began. Queries made by a transaction running in serializable isolation level can access changes made by the transaction itself. Transaction-level read consistency results in repeatable reads and protects a query from phantom anomalies.

2.2.1 Transaction lifecycle

As it is described in [1, 15]:

At the beginning of a transaction, an available undo data segment is assigned to the transaction in order to record the necessary undo entries. A *transaction ID* is not allocated until an undo segment is allocated. This will take place during the first DML statement. A transaction ID is unique to each transaction and represents the undo segment number. Also, Oracle makes a note of the current System Change Number (SCN).

A system change number (SCN) is the internal timestamp used by Oracle RDBMS. Its primary purpose is to order events that take place. This is a necessary tool to make sure that the ACID properties of a transaction are satisfied. SCNs occur in a monotonically increasing sequence. The SCN indicates a logical point in time and because of that it can be used by the RDBMS like a clock. When an event has a smaller SCN than another, it means that it occurred at an earlier point in time. Events that occurred at the same time within the database will share the same SCN. Every transaction is assigned its own SCN. For example, if a transaction updates a row, then the database records the SCN at which this update occurred. Other modifications in this transaction have the same SCN.

When reading a table, Oracle determines, with the use of the SCN number, if the page contains the effects of other concurrent transactions that should not be visible to the current transaction, depending on the isolation level in use. In case the page is found to contain effects of concurrent transactions, then an older version of the page is reconstructed by undoing the effects of each related transaction. It parses the undo records associated

with each transaction and applies them to the page until the effects of those transactions are removed. The final page constructed in this way is finally used to access the tuples within it.

In Figure 2.1 we see that the transaction with SCN 10023 is performing a `SELECT` statement. The rows that match the `SELECT` statement's condition but have a higher SCN get replaced by their previous versions (those with lower SCN than the current transaction's) that were stored in the Undo Segment.

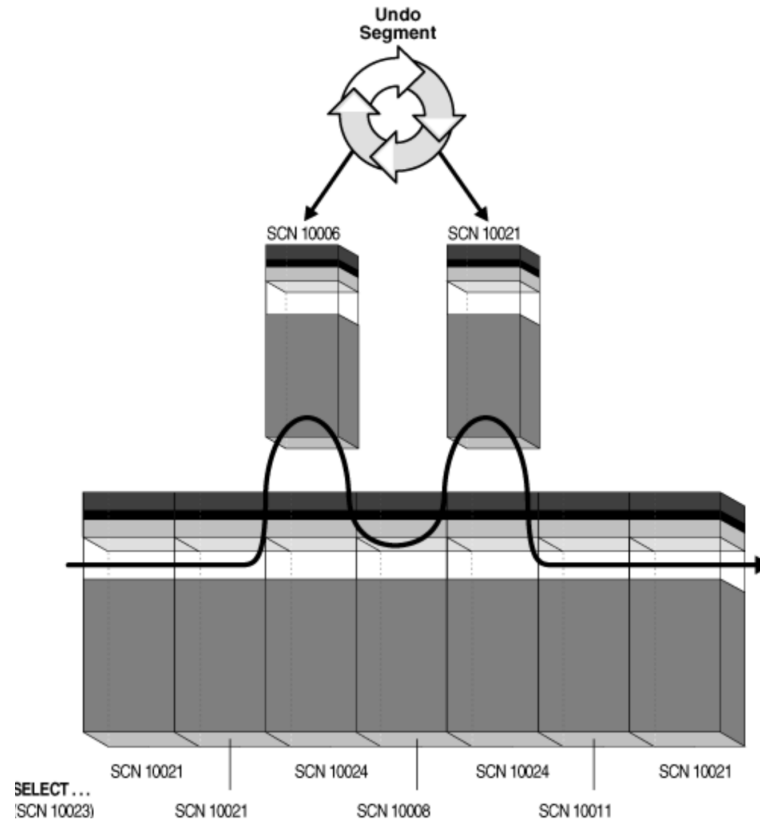


Figure 2.1: Read consistency in the READ COMMITTED isolation level[1]

When a transaction modifies data, Oracle RDBMS copies the original data before modifying it in an **Undo Segment**. The original copy of the modified data is called undo data. Storing this information is crucial for the following reasons:

- To undo any uncommitted changes made to the database when a rollback is issued. A rollback can be needed if a user wants to undo the changes of an erroneous or unintended transaction, or it can be part of a recovery operation.
- To ensure read consistency. Meaning that each user can get a consistent view of data, even while other changes may be occurring against that data. A user session will not access uncommitted changes made in other user sessions. For example, if a user issues a query at 9:00 p.m. and the query lasts 9 minutes, then the query results reflect the entire state of the data at 9:00 p.m., regardless of update or insert operations performed by other users after the query started.

Until the transaction is committed or rolled back, changes in the data are considered temporary. Before the end of the transaction, the state of the data is as follows:

- Undo data are created. The undo data contain the previous data values that got changed by the SQL statements of the current transaction.
- The rows affected by the data change are locked. Other users cannot change the data in the affected rows, nor can they see the uncommitted changes.

When a transaction commits, the following actions occur:

- An SCN is generated for the COMMIT.
- The fact that the transaction has committed gets recorded in the internal transaction table for the associated undo tablespace. The related unique SCN of the transaction is assigned and recorded in the transaction table.
- Locks held on rows and tables get released. Users who were waiting on locks held by the uncommitted transaction are allowed to proceed.
- Transaction gets marked as completed.

After a transaction commits, users can view the applied changes.

When rolling back an entire transaction, the following actions are performed:

- All the SQL statements in the transaction get undone, by using the corresponding undo segments.
- The transaction table entry for every active transaction contains a pointer to all the undo data for the transaction.
- The database reads the data from the undo segment, reverses the operation(s), and then marks the undo entry as applied. Thus:
 - If a transaction inserts a row, then a rollback deletes it.
 - If a transaction updates a row, then a rollback reverses the update.
 - If a transaction deletes a row, then a rollback reinserts it.
- All the locks held by the transaction get released.
- Ends the transaction.

2.3 Examples of isolation levels

The following scenarios are being tested in two, side-by-side active concurrent transactions. The table that is being used got created with this SQL code:

```
CREATE TABLE T (id INT NOT NULL PRIMARY KEY, x INT);
INSERT INTO T VALUES (1, 1);
INSERT INTO T VALUES (2, 2);
INSERT INTO T VALUES (3, 3);
```

2.3.1 Non-repeatable read and READ COMMITTED

SESSION A

```
ORACLE - TEST1 - # Anomaly|Non-Repeatable Read - Isolation|Read Committed
File Edit View Terminal Tabs Help
SQL> SQL> -- S T E P 1 --
SQL> -----
SQL> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
Transaction set.
SQL> SELECT * FROM T WHERE id=1;
      ID      X
-----
      1      1
SQL> -- S T E P 3 --
SQL> -----
SQL> SELECT * FROM T ORDER BY id;
      ID      X
-----
      1      1
      2      2
      3      3
SQL> -- S T E P 5 --
SQL> -----
SQL> SELECT * FROM T ORDER BY id;
      ID      X
-----
      1     12
      2      2
      3      3
SQL> COMMIT;
Commit complete.
SQL> █
```

SESSION B

```
SQL>
SQL> -- S T E P 2 --
SQL> -----
SQL> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
Transaction set.
SQL> UPDATE T SET x=12 WHERE id=1;
1 row updated.
SQL> SELECT * FROM T WHERE id=1;
      ID      X
-----
      1     12
SQL> -- S T E P 4 --
SQL> -----
SQL> COMMIT;
Commit complete.
SQL>
```

Figure 2.2: Non-repeatable read in Oracle's READ COMMITTED isolation level

Here one can see that Oracle's READ COMMITTED isolation level does not provide a repeatable read, as Step 5 of Session A shows the changed value that Session B committed in Step 4. Also, a dirty read is not allowed as Step 3 of Session A would be able to see the changed value if that was the case.

2.3.2 Non-repeatable read and SERIALIZABLE

SESSION A

SESSION B

```

ORACLE - TEST2 - # Anomaly|Non-Repeatable Read - Isolation|Serializable

File Edit View Terminal Tabs Help

SQL>
SQL> -- S T E P 1 --
SQL> -----
SQL> SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
Transaction set.
SQL> SELECT * FROM T WHERE id=1;

   ID      X
-----
    1      1

SQL> -- S T E P 3 --
SQL> -----
SQL> SELECT * FROM T WHERE id=1;

   ID      X
-----
    1      1

SQL> COMMIT;
Commit complete.
SQL>

SQL>
SQL> -- S T E P 2 --
SQL> -----
SQL> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
Transaction set.
SQL> UPDATE T SET x=12 WHERE id=1;
1 row updated.
SQL> SELECT * FROM T WHERE id=1;

   ID      X
-----
    1     12

SQL> COMMIT;
Commit complete.
SQL>

```

Figure 2.3: Non-repeatable read in Oracle's SERIALIZABLE isolation level

The SERIALIZABLE isolation level provides repeatable read functionality.

One can see that Session A on Step 3 sees the table value unchanged, even after Session B has committed on Step 2.

2.3.3 Lost update and READ COMMITTED

SESSION A

SESSION B

```

ORACLE - TEST3 - # Anomaly|Lost Update - Isolation|Read Committed
File Edit View Terminal Tabs Help

SQL>
SQL> -- STEP 1 --
SQL> -----
SQL> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
Transaction set.
SQL> SELECT * FROM T WHERE id=1;
      ID      X
-----
      1      1

SQL> -- STEP 3 --
SQL> -----
SQL> UPDATE T SET x=13 WHERE id=1;
1 row updated.
SQL> SELECT * FROM T WHERE id=1;
      ID      X
-----
      1     13

SQL> COMMIT;
Commit complete.
SQL>

SQL>
SQL> -- STEP 2 --
SQL> -----
SQL> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
Transaction set.
SQL> UPDATE T SET x=12 WHERE id=1;
1 row updated.
SQL> SELECT * FROM T WHERE id=1;
      ID      X
-----
      1     12

SQL> COMMIT;
Commit complete.
SQL>

```

Figure 2.4: Lost update in Oracle's READ COMMITTED isolation level

Here one sees the lost update phenomenon occurring in the READ COMMITTED isolation level. Session B's update ($x=12$, on Step 2) gets overwritten by Session A's update ($x=13$, on Step 3), without a warning or an exception, which would have been the case in the SERIALIZABLE isolation level.

2.3.4 Lost update and SERIALIZABLE

SESSION A

SESSION B

```

ORACLE - TEST4 - # Anomaly|Lost Update - Isolation|Serializable

File Edit View Terminal Tabs Help

SQL>
SQL> -- STEP 1 --
SQL> -----
SQL> SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
Transaction set.
SQL> SELECT * FROM T WHERE id=1;

   ID      X
-----
    1      1

SQL> -- STEP 3 --
SQL> -----
SQL> UPDATE T SET x=13 WHERE id=1;
UPDATE T SET x=13 WHERE id=1
*
ERROR at line 1:
ORA-08177: can't serialize access for this transaction

SQL> COMMIT;
Commit complete.

SQL> SELECT * FROM T WHERE id=1;

   ID      X
-----
    1     12

SQL> COMMIT;
Commit complete.

SQL>

```

```

SQL>
SQL> -- STEP 2 --
SQL> -----
SQL> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
Transaction set.
SQL> UPDATE T SET x=12 WHERE id=1;
1 row updated.
SQL> SELECT * FROM T WHERE id=1;

   ID      X
-----
    1     12

SQL> COMMIT;
Commit complete.

SQL>

```

Figure 2.5: Lost update in Oracle's SERIALIZABLE isolation level

In a SERIALIZABLE isolation environment, an exception is thrown the moment a lost update phenomenon is about to occur. This can be seen in Session A, where Step 3 tries to update a tuple that got updated and committed by Session B on Step 2. An exception *"Can't serialize access for this transaction"* is thrown. Interestingly enough, the transaction continues after the exception, which means that it is the responsibility of the developer to handle it (e.g., rollback instead of commit).

2.3.5 Ghost update and READ COMMITTED

SESSION A

SESSION B

```

ORACLE - TEST5 - # Anomaly|Ghost Update - Isolation|Read Committed

File Edit View Terminal Tabs Help

SQL>
SQL> -- STEP 1 --
SQL> -----
SQL> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
Transaction set.
SQL> SELECT id,x FROM T ORDER BY id;
   ID      X
-----
    1      1
    2      2
    3      3

SQL> -- STEP 3 --
SQL> -----
SQL> UPDATE T SET x=12 WHERE id=1;
0 rows updated.

SQL> COMMIT;
Commit complete.

SQL> SELECT id,x FROM T ORDER BY id;
   ID      X
-----
    2      2
    3      3

SQL> COMMIT;
Commit complete.

SQL>

SQL>
SQL> -- STEP 2 --
SQL> -----
SQL> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
Transaction set.
SQL> DELETE FROM T WHERE id=1;
1 row deleted.

SQL> SELECT id,x FROM T ORDER BY id;
   ID      X
-----
    2      2
    3      3

SQL> COMMIT;
Commit complete.

SQL>

```

Figure 2.6: Ghost update in Oracle's READ COMMITTED isolation level

The ghost update, that was bound to happen, in Session A on Step 3, did not occur and the `UPDATE` query resulted in *"0 rows updated"*. This is because it was previously deleted by Session B on Step 2. Notably, no exception or rollback took place but the transaction continued. Ghost update phenomena do not happen in Oracle's READ COMMITTED isolation level.

2.3.6 Ghost update and SERIALIZABLE

SESSION A

SESSION B

```

ORACLE - TEST6 - # Anomaly|Ghost Update - Isolation|Serializable

File Edit View Terminal Tabs Help

SQL>
SQL> -- S T E P 1 --
SQL> -----
SQL> SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
Transaction set.
SQL> SELECT id,x FROM T ORDER BY id;
      ID      X
-----
      1      1
      2      2
      3      3
SQL> -- S T E P 3 --
SQL> -----
SQL> UPDATE T SET x=12 WHERE id=1;
UPDATE T SET x=12 WHERE id=1
*
ERROR at line 1:
ORA-08177: can't serialize access for this transaction

SQL> COMMIT;
Commit complete.

SQL> SELECT id,x FROM T ORDER BY id;
      ID      X
-----
      2      2
      3      3
SQL> COMMIT;
Commit complete.
SQL>

SQL>
SQL> -- S T E P 2 --
SQL> -----
SQL> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
Transaction set.
SQL> DELETE FROM T WHERE id=1;
1 row deleted.
SQL> SELECT id,x FROM T ORDER BY id;
      ID      X
-----
      2      2
      3      3
SQL> COMMIT;
Commit complete.
SQL>

```

Figure 2.7: Ghost update in Oracle's SERIALIZABLE isolation level

The REPEATABLE READ isolation level is prone to ghost update anomalies. Oracle's SERIALIZABLE prevents them from happening by stopping the execution with a message **can't serialize access for this transaction** (Session A Step 3). No automatic rollback is performed, which means that the developer must handle the exception appropriately by, for example, retrying the query or rolling back the whole transaction.

2.3.7 Phantom update and READ COMMITTED

SESSION A

SESSION B

The screenshot shows an Oracle SQL*Plus window titled "ORACLE - TEST7 - # Anomaly|Phantom Update - Isolation|Read Committed". It is split into two panes for Session A and Session B.

Session A:

```
SQL>
SQL> -- Step 1. Session 1, Transaction T1
SQL> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
Transaction set.
SQL> SELECT id,x FROM T ORDER BY id;
-----
ID      X
-----
1        1
2        2
3        3
SQL> -- Step 3. T1 of Session 1 continues
SQL> UPDATE T SET x=42 WHERE id=4;
1 row updated.
SQL> SELECT id,x FROM T ORDER BY id;
-----
ID      X
-----
1        1
2        2
3        3
4       42
SQL> COMMIT;
Commit complete.
SQL> SELECT id,x FROM T ORDER BY id;
-----
ID      X
-----
1        1
2        2
3        3
4       42
SQL> COMMIT;
Commit complete.
SQL>
```

Session B:

```
SQL>
SQL> -- Step 2. Meanwhile in a concurrent Session 2, Transaction 2
SQL> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
Transaction set.
SQL> INSERT INTO T VALUES (4,4);
1 row created.
SQL> COMMIT;
Commit complete.
SQL>
```

Figure 2.8: Phantom update in Oracle's READ COMMITTED isolation level

Phantom updates are not prevented in Oracle's READ COMMITTED isolation level. This can be seen in Session A. Step 3 successfully updates the new row that got inserted (and committed) by Session B on Step 2.

2.3.8 Phantom update and SERIALIZABLE

SESSION A

SESSION B

```

ORACLE - TEST8 - # Anomaly|Phantom Update - Isolation|Serializable

File Edit View Terminal Tabs Help

SQL>
SQL> -- S T E P 1 --
SQL> -----
SQL> SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
Transaction set.
SQL> SELECT id,x FROM T ORDER BY id;
   ID      X
-----
    1      1
    2      2
    3      3

SQL> -- S T E P 3 --
SQL> -----
SQL> UPDATE T SET x=42 WHERE id=4;
0 rows updated.
SQL> SELECT id,x FROM T ORDER BY id;
   ID      X
-----
    1      1
    2      2
    3      3

SQL> COMMIT;
Commit complete.
SQL> SELECT id,x FROM T ORDER BY id;
   ID      X
-----
    1      1
    2      2
    3      3
    4      4

SQL> COMMIT;
Commit complete.
SQL> █

SQL>
SQL> -- S T E P 2 --
SQL> -----
SQL> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
Transaction set.
SQL> INSERT INTO T VALUES (4,4);
1 row created.
SQL> COMMIT;
Commit complete.
SQL>

```

Figure 2.9: Phantom update in Oracle's SERIALIZABLE isolation level

In the case of phantom updates, the SERIALIZABLE isolation, because of its "repeatable read" functionality, prevents such phenomena.

As one can see in Session A on Step 3, if one attempts to update a tuple that did not exist in the start of a transaction, but got inserted and committed in another transaction (Session B Step 2) while the first is still active, he will be presented with "*0 rows updated.*" message. This means that Oracle's SERIALIZABLE isolation level prevents phantom updates without exceptions or automatic rollbacks.

2.3.9 Read skew and READ COMMITTED

Read skew anomaly is going to get examined using the following table which is a simulation of two bank accounts belonging to a user. The user is attempting to transfer money from one account to the other.

```
CREATE TABLE accounts (account_id INT NOT NULL PRIMARY KEY, balance INT);
INSERT INTO accounts VALUES (1, 500);
INSERT INTO accounts VALUES (2, 500);
```

SESSION A

SESSION B

```

ORACLE - TEST9 - # Anomaly|Read Skew - Isolation|Read Committed

File Edit View Terminal Tabs Help

1 row created.
SQL> INSERT INTO accounts VALUES (2, 500);
1 row created.
SQL> COMMIT;
Commit complete.
SQL> ---- S T E P 1 ----
SQL> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
Transaction set.
SQL> SELECT * FROM accounts;
ACCOUNT_ID  BALANCE
-----
1           500
2           500
SQL> ---- S T E P 3 ----
SQL> UPDATE accounts SET balance = balance - 100 WHERE account_id = 1;
1 row updated.
SQL> SELECT * FROM accounts;
ACCOUNT_ID  BALANCE
-----
1           400
2           500
SQL> ---- S T E P 5 ----
SQL> UPDATE accounts SET balance = balance + 100 WHERE account_id = 2;
1 row updated.
SQL> COMMIT;
Commit complete.
SQL> SELECT * from accounts;
ACCOUNT_ID  BALANCE
-----
1           400
2           600
SQL>

SQL> ---- S T E P 2 ----
SQL> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
Transaction set.
SQL> SELECT * FROM accounts;
ACCOUNT_ID  BALANCE
-----
1           500
2           500
SQL> ---- S T E P 4 ----
SQL> SELECT balance from accounts WHERE account_id = 2;
BALANCE
-----
500
SQL> ---- S T E P 6 ----
SQL> SELECT balance from accounts WHERE account_id = 1;
BALANCE
-----
400
SQL>

```

Figure 2.10: Read skew in Oracle's READ COMMITTED isolation level

In Steps 3 and 5 of the left session (Session A) of Figure 2.10, an attempt is made to transfer **€100** from account 1 to account 2. In Step 4 of the session in the right (Session B) of Figure 2.10, the current balance of account 2 is read before Step 5 (of Session A) adds **€100** to it. The value read equals **€500**. Next, Step 6 of Session B reads the current balance of account 1, following the subtraction of **€100** in Step 3 of Session A. The value read is **€400**. As a result, the transaction in Session B has read a combined value of accounts 1 and 2 that

is equal to $500 + 400 = \text{€}900$ instead of the correct value which is **€1000**. This comprises the so called read skew problem. In essence, it is a violation of the data integrity constraint requiring the combined value of the two bank accounts to always remain equal to **€1000**. Oracle's READ COMMITTED isolation level is prone to Read Skew anomalies.

2.3.10 Read skew and SERIALIZABLE

SESSION A

SESSION B

```

ORACLE - TEST10 - # Anomaly|Read Skew - Isolation|Serializable
File Edit View Terminal Tabs Help

1 row created.
SQL> INSERT INTO accounts VALUES (2, 500);
1 row created.
SQL> COMMIT;
Commit complete.
SQL> ---- S T E P 1 ----
SQL> SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
Transaction set.
SQL> SELECT * FROM accounts;
ACCOUNT_ID  BALANCE
-----
1           500
2           500
SQL> ---- S T E P 3 ----
SQL> UPDATE accounts SET balance = balance - 100 WHERE account_id = 1;
1 row updated.
SQL> SELECT * FROM accounts;
ACCOUNT_ID  BALANCE
-----
1           400
2           500
SQL> ---- S T E P 5 ----
SQL> UPDATE accounts SET balance = balance + 100 WHERE account_id = 2;
1 row updated.
SQL> COMMIT;
Commit complete.
SQL> SELECT * from accounts;
ACCOUNT_ID  BALANCE
-----
1           400
2           600
SQL>

SQL> ---- S T E P 2 ----
SQL> SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
Transaction set.
SQL> SELECT * FROM accounts;
ACCOUNT_ID  BALANCE
-----
1           500
2           500
SQL> ---- S T E P 4 ----
SQL> SELECT balance from accounts WHERE account_id = 2;
BALANCE
-----
500
SQL> ---- S T E P 6 ----
SQL> SELECT balance from accounts WHERE account_id = 1;
BALANCE
-----
500
SQL>

```

Figure 2.11: Read skew in Oracle's SERIALIZABLE isolation level

Oracle's Serializable isolation level, due to its repeatable read functionality, is not prone to read skew anomalies. So no matter *when* Session B reads each bank account, it will consistently read 500 for each one of them.

2.3.11 Write skew and SERIALIZABLE

To examine the write skew anomaly, the SQL code below is used.

The code creates a hospital table with doctors and their "on call" status.

Note: The hospital database model has a constraint requiring that **at least one** doctor must always be on call.

```
CREATE TABLE doctors (name VARCHAR2(255) NOT NULL PRIMARY KEY,
                      on_call VARCHAR2(10), shift_id INT);
INSERT INTO doctors (name, on_call, shift_id)
VALUES ('Alice', 'true', 1234);
INSERT INTO doctors (name, on_call, shift_id)
VALUES ('Bob', 'true', 1234);
INSERT INTO doctors (name, on_call, shift_id)
VALUES ('Carol', 'false', 1234);
CREATE PROCEDURE currently_on_call (doctors_on_call OUT number) IS
BEGIN
    SELECT count(*) INTO doctors_on_call
    FROM doctors WHERE on_call='true' AND shift_id=1234;
END
```

Figure 2.12 below illustrates a case where Alice and Bob apply for a leave simultaneously.

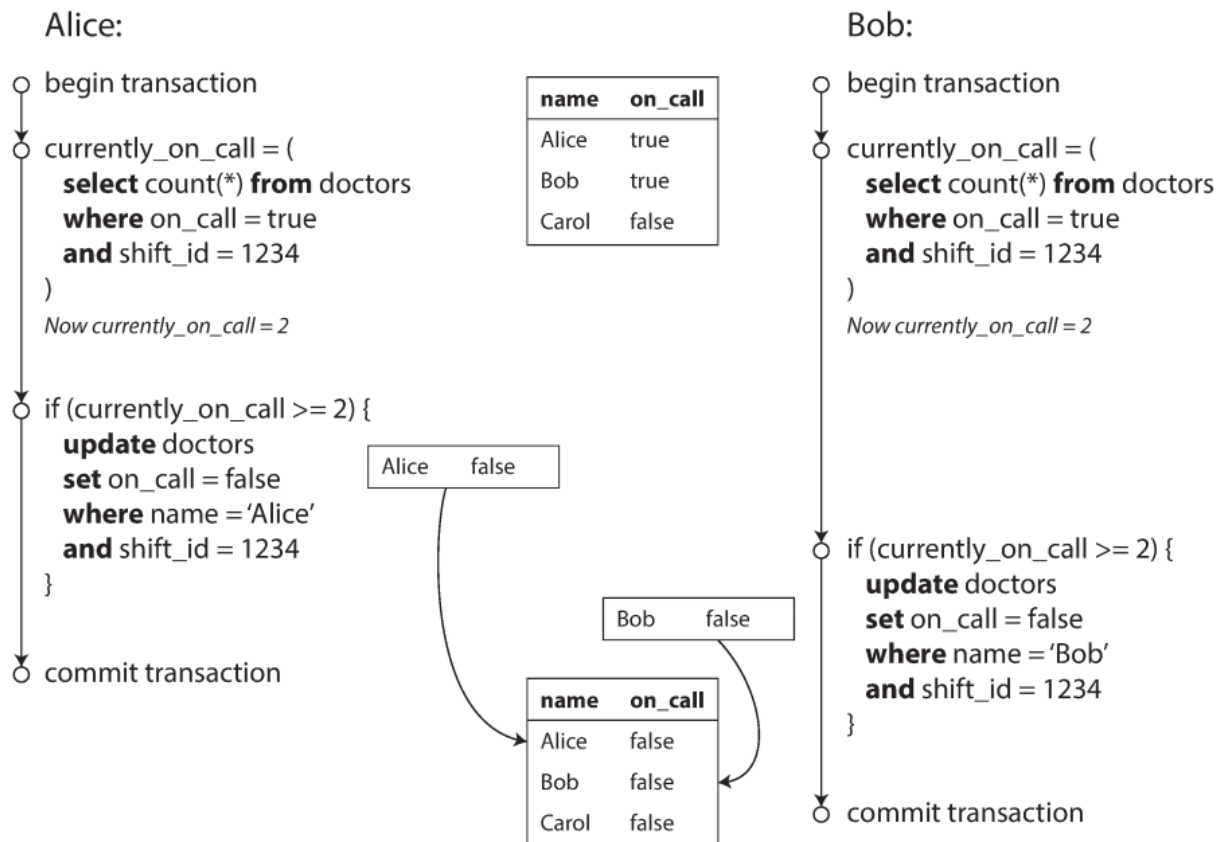


Figure 2.12: Example of write skew occurring (all false) [2]

SESSION A

SESSION B

```

ORACLE - TEST12 - # Anomaly|Write Skew - Isolation|Serializable

File Edit View Terminal Tabs Help

SQL> -- S T E P 1 --
SQL> -----
SQL> SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

Transaction set.

SQL> SELECT * FROM doctors;

NAME      ON_CALL SHIFT_ID
-----
Alice     true    1234
Bob       true    1234
Carol     false   1234

SQL> var doctors_on_call number;
SQL> exec currently_on_call(:doctors_on_call);

PL/SQL procedure successfully completed.

SQL> print :doctors_on_call;

DOCTORS_ON_CALL
-----
                2

SQL> -- S T E P 3 --
SQL> -----
SQL> UPDATE doctors SET on_call = CASE WHEN :doctors_on_call >= 2 THEN 'false' ELSE 'true' END WHERE name = 'Alice' and shift_id=1234;

1 row updated.

SQL> SELECT * FROM doctors;

NAME      ON_CALL SHIFT_ID
-----
Alice     false   1234
Bob       true    1234
Carol     false   1234

SQL> COMMIT;

Commit complete.

SQL> -- S T E P 5 --
SQL> -----
SQL> SELECT * FROM doctors;

NAME      ON_CALL SHIFT_ID
-----
Alice     false   1234
Bob       true    1234
Carol     false   1234

SQL> COMMIT;

Commit complete.

SQL> clear screen
-- S T E P 2 --
SQL> SQL> -----
SQL> SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

Transaction set.

SQL> SELECT * FROM doctors;

NAME      ON_CALL SHIFT_ID
-----
Alice     true    1234
Bob       true    1234
Carol     false   1234

SQL> -- S T E P 4 --
SQL> -----
SQL> var doctors_on_call number;
SQL> exec currently_on_call(:doctors_on_call);

PL/SQL procedure successfully completed.

SQL> print :doctors_on_call;

DOCTORS_ON_CALL
-----
                2

SQL> UPDATE doctors SET on_call = CASE WHEN :doctors_on_call >= 2 THEN 'false' ELSE 'true' END WHERE name = 'Bob' and shift_id=1234;
UPDATE doctors SET on_call = CASE WHEN :doctors_on_call >= 2 THEN 'false' ELSE 'true' END WHERE name = 'Bob' and shift_id=1234
*
ERROR at line 1:
ORA-08177: can't serialize access for this transaction

SQL> COMMIT;

Commit complete.

SQL> SELECT * FROM doctors;

NAME      ON_CALL SHIFT_ID
-----
Alice     false   1234
Bob       true    1234
Carol     false   1234

SQL> -- S T E P 6 --
SQL> -----
SQL> SELECT * FROM doctors;

NAME      ON_CALL SHIFT_ID
-----
Alice     false   1234
Bob       true    1234
Carol     false   1234

```

Figure 2.13: Write skew in Oracle's SERIALIZABLE isolation level

Write skew anomalies are typically present in the REPEATABLE READ isolation level. Oracle's SERIALIZABLE isolation level, can be compared to other Database Management Systems' REPEATABLE READ isolation level.

As one can see, in Step 4 of Session B, even though the doctors on call seem to be 2, the update is not allowed and a *"can't serialize access for this transaction"* exception is thrown. Therefore, write skew anomalies are prevented in Oracle's SERIALIZABLE isolation level, but the developer has to handle the exception appropriately.

2.3.12 Write skew and READ COMMITTED

SESSION A

```

ORACLE - TEST11 - # Anomaly|Write Skew - Isolation|Read Committed

File Edit View Terminal Tabs Help

SQL> -- S T E P 1 --
SQL> -----
SQL> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

Transaction set.

SQL> SELECT * FROM doctors;

NAME      ON_CALL SHIFT_ID
-----
Alice     true    1234
Bob       true    1234
Carol     false   1234

SQL> var doctors_on_call number;
SQL> exec currently_on_call(:doctors_on_call);

PL/SQL procedure successfully completed.

SQL> print :doctors_on_call;

DOCTORS_ON_CALL
-----
                2

SQL> -- S T E P 3 --
SQL> -----
SQL> UPDATE doctors SET on_call = CASE WHEN :doctors_on_call >= 2 THEN 'false' ELSE 'true' END WHERE name = 'Alice' and shift_id=1234;

1 row updated.

SQL> SELECT * FROM doctors;

NAME      ON_CALL SHIFT_ID
-----
Alice     false   1234
Bob       true    1234
Carol     false   1234

SQL> COMMIT;

Commit complete.

SQL> -- S T E P 5 --
SQL> -----
SQL> SELECT * FROM doctors;

NAME      ON_CALL SHIFT_ID
-----
Alice     false   1234
Bob       true    1234
Carol     false   1234

SQL> COMMIT;

Commit complete.

SQL>

SQL> -- S T E P 2 --
SQL> -----
SQL> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

Transaction set.

SQL> SELECT * FROM doctors;

NAME      ON_CALL SHIFT_ID
-----
Alice     true    1234
Bob       true    1234
Carol     false   1234

SQL> -- S T E P 4 --
SQL> -----
SQL> var doctors_on_call number;
SQL> exec currently_on_call(:doctors_on_call);

PL/SQL procedure successfully completed.

SQL> print :doctors_on_call;

DOCTORS_ON_CALL
-----
                1

SQL> UPDATE doctors SET on_call = CASE WHEN :doctors_on_call >= 2 THEN 'false' ELSE 'true' END WHERE name = 'Bob' and shift_id=1234;

1 row updated.

SQL> COMMIT;

Commit complete.

SQL> SELECT * FROM doctors;

NAME      ON_CALL SHIFT_ID
-----
Alice     false   1234
Bob       true    1234
Carol     false   1234

SQL> -- S T E P 6 --
SQL> -----
SQL> SELECT * FROM doctors;

NAME      ON_CALL SHIFT_ID
-----
Alice     false   1234
Bob       true    1234
Carol     false   1234

SQL> COMMIT;

Commit complete.

SQL>

```

Figure 2.14: Write skew in Oracle's READ COMMITTED isolation level

As expected, write skew phenomena are not encountered in the READ COMMITTED isolation level. So, one can see in Session B, that Step 4 correctly sees 1 available doctor on call and as a result Bob is not allowed to take his absence leave, because the hospital must have at least 1 doctor on call.

2.4 Comparison of test results

This table sums up the findings of how Oracle isolation levels behave when encountering the tested transaction anomalies.

Table 2.1: Oracle isolation levels and transaction anomalies

	Read Committed	Serializable
NON-REPEATABLE READ	Possible	Not Possible
LOST UPDATE	Possible	Not Possible ¹
GHOST UPDATE	Not Possible	Not Possible ¹
PHANTOM UPDATE	Possible	Not Possible ²
READ SKEW	Possible	Not Possible
WRITE SKEW	Not Possible	Not Possible ¹

¹ Exception occurs and must be handled programmatically. Otherwise, the transaction continues.

² Exception does not occur.

Chapter 3

MVCC in MySQL

3.1 MySQL Transaction Isolation Levels

The most widely used storage engine for MySQL that supports transactions is InnoDB.

A storage engine is a software module that a Database Management System uses to create, read, update data from a database [16].

InnoDB is going to be used as MySQL's storage engine for the purposes of this thesis. InnoDB, as of version 10.3.15, supports the following transaction isolation levels, all of which are described by the SQL:1992 standard:

- READ UNCOMMITTED
- READ COMMITTED
- REPEATABLE READ
- SERIALIZABLE

Multi-versioning, or *consistent read*, as MySQL describes it, is implemented in the READ COMMITTED and the REPEATABLE READ isolation levels [17], so only those two fall into the scope of this thesis.

A *consistent read* does not acquire locks on the tables that needs access to. Therefore, other sessions can modify those tables at the same time a consistent read is being performed on the table [17].

3.1.1 READ COMMITTED

The READ COMMITTED transaction isolation level, in MySQL, is *not* the default option. This is the opposite of what happens in other Database Management Systems. In the READ COMMITTED isolation level a fresh snapshot is being obtained for every query within each transaction.

As described in [18]:

For *locking* reads (SELECT with FOR UPDATE or FOR SHARE), UPDATE statements, and DELETE statements, MySQL locks only index records, not the gaps before them, and as a result, new records can be inserted next to the locked records. Because of this, phantom problems may occur.

UPDATE and DELETE statements hold locks only on the rows that are about to be updated/deleted. Non-matching rows get their locks released after MySQL has evaluated the WHERE condition.

For UPDATE statements, if a row is already locked, MySQL returns the latest committed version, so that it can be determined whether the row matches the UPDATE's WHERE clause. In the case of a match, meaning that a row must be updated, MySQL reads the row one more time and this time it either locks it or waits to acquire a lock on it.

3.1.2 REPEATABLE READ

This is MySQL's default transaction isolation level. All consistent reads within a transaction, read the snapshot established at the beginning of the transaction, allowing for a consistent view of the database from statement to statement. To get a more fresh snapshot for your queries, commit the current transaction and issue new queries afterwards.

As per [18]:

The snapshot of the database state applies to `SELECT` statements within a transaction, not necessarily to DML statements. If you insert or update some rows and then commit that transaction, a `DELETE` or `UPDATE` statement issued from another concurrent transaction could affect those committed rows, even though the session could not query them at the beginning. If a transaction updates or deletes rows that got committed by a different transaction, those changes do become visible to the current transaction.

This isolation level, provides repeatable reads and protects against phantom reads and ghost updates, but it is, surprisingly, susceptible to phantom updates and lost updates. It seems as MySQL's `REPEATABLE READ` is less strict than other Database Management Systems' similar isolation levels.

3.2 MySQL and MVCC

MySQL implements multi-version concurrency control (MVCC), which means that each user will access their own set of data (snapshot) to interact with. This allows users to see a consistent view of the system without expensive and performance-constraining locking which would limit concurrency.

A primary feature of MySQL's implementation of MVCC is that when a record is modified, its current version is first stashed away as an "undo record" in an "undo log". It is named "undo log" because it keeps the necessary information to undo the applied changes, so as to revert the record to its previous version.

Every record has a reference to its most recent undo record. This is called a **rollback pointer**, and every undo record contains a reference to its previous undo record (except for an initial record insert, which can be undone by simply deleting the record), forming a chain of all previous versions of a record. In this way, any previous version of a record can be easily re-constructed [3].

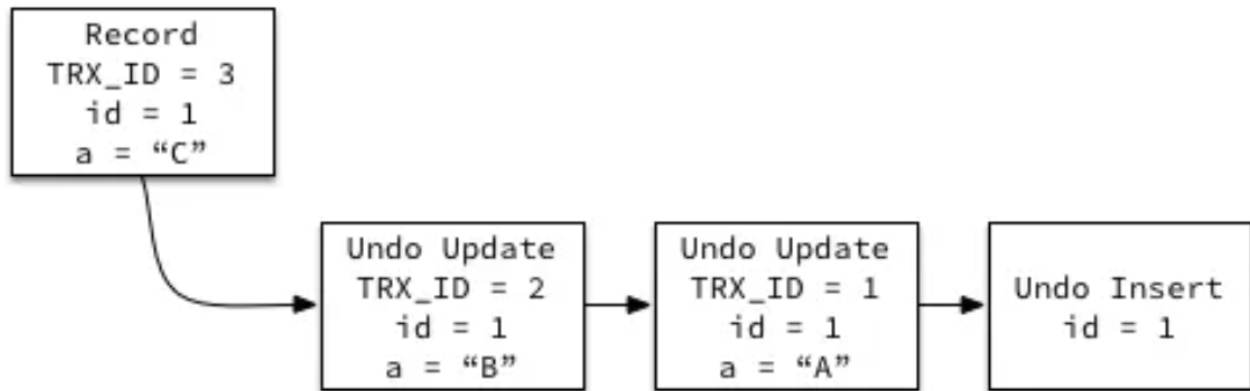


Figure 3.1: MySQL Undo Log chaining representation [3]

[19] mentions:

Undo logs in the rollback segment are separated into *insert* and *update* undo logs. Insert undo logs are needed only when rolling back a transaction and can be discarded as soon as the transaction commits. Update undo logs are used also in consistent reads. They can be discarded only if there is no transaction present for which MySQL has assigned a snapshot that a consistent read could need the information to build an earlier version of a database row.

MySQL, as described in [19], among other things, internally adds three necessary fields to each row so as to facilitate MVCC:

- A 6-byte `DB_TRX_ID` field which indicates the transaction identifier for the last transaction that inserted or modified the row. These ids are incremented in a way that allows MySQL to determine that one transaction started before another. So, along with the columns that may have been updated, a write operation also marks the row with its transaction id. This allows read operations to know if they can use the row data, or if it has been changed and an older version needs to be consulted.
- A 7-byte `DB_ROLL_PTR` field, called the roll pointer, which points to an undo log record written to the rollback segment.
- A 6-byte `DB_ROW_ID` field which contains an ID that increments as new rows are inserted.

3.3 Examples of isolation levels

The following scenarios are being tested in two, side-by-side active concurrent transactions. The table that is being used got created with this SQL code:

```
CREATE TABLE T (id INT NOT NULL PRIMARY KEY, x INT);
INSERT INTO T VALUES (1, 1);
INSERT INTO T VALUES (2, 2);
INSERT INTO T VALUES (3, 3);
```

3.3.1 Non-repeatable read and READ COMMITTED

SESSION A

SESSION B

```

MYSQL - TEST1 - # Anomaly|Non-Repeatable Read - Isolation|Read Committed

File Edit View Terminal Tabs Help

MariaDB [testdb]>
MariaDB [testdb]> -- STEP 1 --
MariaDB [testdb]> -----
MariaDB [testdb]> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> BEGIN;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> SELECT * FROM T WHERE id=1;
+-----+
| id | x |
+-----+
| 1 | 1 |
+-----+
1 row in set (0.000 sec)

MariaDB [testdb]> -- STEP 3 --
MariaDB [testdb]> -----
MariaDB [testdb]> SELECT * FROM T ORDER BY id;
+-----+
| id | x |
+-----+
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
+-----+
3 rows in set (0.000 sec)

MariaDB [testdb]> -- STEP 5 --
MariaDB [testdb]> -----
MariaDB [testdb]> SELECT * FROM T ORDER BY id;
+-----+
| id | x |
+-----+
| 1 | 12 |
| 2 | 2 |
| 3 | 3 |
+-----+
3 rows in set (0.000 sec)

MariaDB [testdb]> COMMIT;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]>

MariaDB [testdb]>
MariaDB [testdb]> -- STEP 2 --
MariaDB [testdb]> -----
MariaDB [testdb]> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> BEGIN;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> UPDATE T SET x=12 WHERE id=1;
Query OK, 1 row affected (0.004 sec)
Rows matched: 1 Changed: 1 Warnings: 0

MariaDB [testdb]> SELECT * FROM T WHERE id=1;
+-----+
| id | x |
+-----+
| 1 | 12 |
+-----+
1 row in set (0.000 sec)

MariaDB [testdb]> -- STEP 4 --
MariaDB [testdb]> -----
MariaDB [testdb]> COMMIT;
Query OK, 0 rows affected (0.002 sec)

MariaDB [testdb]>

```

Figure 3.2: Non-repeatable read in MySQL's READ COMMITTED isolation level

This example makes clear that MySQL's READ COMMITTED isolation level does not provide a repeatable read, as Session A on Step 3, reads the committed change on Session B. Step 3 on Session A shows that a Dirty Read is avoided as the changed, but not committed, value of Step 2 (Session B) is not seen.

3.3.2 Non-repeatable read and REPEATABLE READ

SESSION A

SESSION B

```

MYSQL - TEST2 - # Anomaly|Non-Repeatable Read - Isolation|Repeatable Read

File Edit View Terminal Tabs Help

MariaDB [testdb]>
MariaDB [testdb]> -- S T E P 1 --
MariaDB [testdb]> -----
MariaDB [testdb]> SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> BEGIN;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> SELECT * FROM T WHERE id=1;
+-----+
| id | x |
+-----+
| 1 | 1 |
+-----+
1 row in set (0.000 sec)

MariaDB [testdb]> -- S T E P 3 --
MariaDB [testdb]> -----
MariaDB [testdb]> SELECT * FROM T WHERE id=1;
+-----+
| id | x |
+-----+
| 1 | 1 |
+-----+
1 row in set (0.000 sec)

MariaDB [testdb]> COMMIT;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> SELECT * FROM T ORDER BY id;
+-----+
| id | x |
+-----+
| 1 | 12 |
| 2 | 2 |
| 3 | 3 |
+-----+
3 rows in set (0.000 sec)

MariaDB [testdb]> COMMIT;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]>

MariaDB [testdb]>
MariaDB [testdb]> -- S T E P 2 --
MariaDB [testdb]> -----
MariaDB [testdb]> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> BEGIN;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> UPDATE T SET x=12 WHERE id=1;
Query OK, 1 row affected (0.000 sec)
Rows matched: 1 Changed: 1 Warnings: 0

MariaDB [testdb]> SELECT * FROM T WHERE id=1;
+-----+
| id | x |
+-----+
| 1 | 12 |
+-----+
1 row in set (0.001 sec)

MariaDB [testdb]> COMMIT;
Query OK, 0 rows affected (0.004 sec)

MariaDB [testdb]>

```

Figure 3.3: Non-repeatable read in MySQL's REPEATABLE READ isolation level

Repeatable read is provided by MySQL's REPEATABLE READ isolation level.

The example above shows that Session A sees Session B's change after committing itself first (Step 3). Before committing, the same value is being read.

3.3.3 Lost update and READ COMMITTED

SESSION A

SESSION B

```

MYSQL - TEST3 - # Anomaly|Lost Update - Isolation|Read Committed

File Edit View Terminal Tabs Help

MariaDB [testdb]>
MariaDB [testdb]> -- STEP 1 --
MariaDB [testdb]> -----
MariaDB [testdb]> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> BEGIN;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> SELECT * FROM T WHERE id=1;
+----+-----+
| id | x |
+----+-----+
| 1 | 1 |
+----+-----+
1 row in set (0.000 sec)

MariaDB [testdb]> -- STEP 3 --
MariaDB [testdb]> -----
MariaDB [testdb]> UPDATE T SET x=13 WHERE id=1;
Query OK, 1 row affected (0.000 sec)
Rows matched: 1 Changed: 1 Warnings: 0

MariaDB [testdb]> SELECT * FROM T WHERE id=1;
+----+-----+
| id | x |
+----+-----+
| 1 | 13 |
+----+-----+
1 row in set (0.000 sec)

MariaDB [testdb]> COMMIT;
Query OK, 0 rows affected (0.004 sec)

MariaDB [testdb]> SELECT * FROM T ORDER BY id;
+----+-----+
| id | x |
+----+-----+
| 1 | 13 |
| 2 | 2 |
| 3 | 3 |
+----+-----+
3 rows in set (0.000 sec)

MariaDB [testdb]> COMMIT;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]>

MariaDB [testdb]>
MariaDB [testdb]> -- STEP 2 --
MariaDB [testdb]> -----
MariaDB [testdb]> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> BEGIN;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> UPDATE T SET x=12 WHERE id=1;
Query OK, 1 row affected (0.000 sec)
Rows matched: 1 Changed: 1 Warnings: 0

MariaDB [testdb]> SELECT * FROM T WHERE id=1;
+----+-----+
| id | x |
+----+-----+
| 1 | 12 |
+----+-----+
1 row in set (0.000 sec)

MariaDB [testdb]> COMMIT;
Query OK, 0 rows affected (0.004 sec)

MariaDB [testdb]>

```

Figure 3.4: Lost update in MySQL's READ COMMITTED isolation level

The READ COMMITTED isolation level is prone to lost update phenomena as one can observe from the above example. Session B's update on Step 2 ($x=12$) gets overwritten by Session A's update on Step 3 ($x=13$) without an exception or a warning.

3.3.4 Lost update and REPEATABLE READ

SESSION A

SESSION B

```

MYSQL - TEST4 - # Anomaly|Lost Update - Isolation|Repeatable Read

File Edit View Terminal Tabs Help

MariaDB [testdb]>
MariaDB [testdb]> -- S T E P 1 --
MariaDB [testdb]> -----
MariaDB [testdb]> SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> BEGIN;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> SELECT * FROM T WHERE id=1;
+-----+
| id | x |
+-----+
| 1 | 1 |
+-----+
1 row in set (0.000 sec)

MariaDB [testdb]> -- S T E P 3 --
MariaDB [testdb]> -----
MariaDB [testdb]> UPDATE T SET x=13 WHERE id=1;
Query OK, 1 row affected (0.000 sec)
Rows matched: 1 Changed: 1 Warnings: 0

MariaDB [testdb]> SELECT * FROM T WHERE id=1;
+-----+
| id | x |
+-----+
| 1 | 13 |
+-----+
1 row in set (0.000 sec)

MariaDB [testdb]> COMMIT;
Query OK, 0 rows affected (0.003 sec)

MariaDB [testdb]> █

MariaDB [testdb]>
MariaDB [testdb]> -- S T E P 2 --
MariaDB [testdb]> -----
MariaDB [testdb]> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> BEGIN;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> UPDATE T SET x=12 WHERE id=1;
Query OK, 1 row affected (0.000 sec)
Rows matched: 1 Changed: 1 Warnings: 0

MariaDB [testdb]> SELECT * FROM T WHERE id=1;
+-----+
| id | x |
+-----+
| 1 | 12 |
+-----+
1 row in set (0.000 sec)

MariaDB [testdb]> COMMIT;
Query OK, 0 rows affected (0.003 sec)

MariaDB [testdb]>

```

Figure 3.5: Lost update in MySQL's REPEATABLE READ isolation level

MySQL's REPEATABLE READ does not prevent lost updates like other DBMS' similar isolation levels do. The example above, shows the Session B's committed value (Step 2) being overwritten by Session A's Step 3 without an exception or a warning.

3.3.5 Ghost update and READ COMMITTED

SESSION A

SESSION B

```

MYSQL - TEST5 - # Anomaly|Ghost Update - Isolation|Read Committed
File Edit View Terminal Tabs Help

MariaDB [testdb]>
MariaDB [testdb]> -- S T E P 1 --
MariaDB [testdb]> -----
MariaDB [testdb]> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> BEGIN;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> SELECT * FROM T WHERE id=1;
+----+-----+
| id | x   |
+----+-----+
| 1  | 1   |
+----+-----+
1 row in set (0.001 sec)

MariaDB [testdb]> -- S T E P 3 --
MariaDB [testdb]> -----
MariaDB [testdb]> UPDATE T SET x=13 WHERE id=1;
Query OK, 0 rows affected (0.001 sec)
Rows matched: 0  Changed: 0  Warnings: 0

MariaDB [testdb]> SELECT * FROM T WHERE id=1;
Empty set (0.000 sec)

MariaDB [testdb]> COMMIT;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> SELECT * FROM T ORDER BY id;
+----+-----+
| id | x   |
+----+-----+
| 2  | 2   |
| 3  | 3   |
+----+-----+
2 rows in set (0.000 sec)

MariaDB [testdb]> COMMIT;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]>

MariaDB [testdb]>
MariaDB [testdb]> -- S T E P 2 --
MariaDB [testdb]> -----
MariaDB [testdb]> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> BEGIN;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> DELETE FROM T WHERE id=1;
Query OK, 1 row affected (0.000 sec)

MariaDB [testdb]> SELECT * FROM T WHERE id=1;
Empty set (0.000 sec)

MariaDB [testdb]> COMMIT;
Query OK, 0 rows affected (0.002 sec)

MariaDB [testdb]>

```

Figure 3.6: Ghost update in MySQL's READ COMMITTED isolation level

Ghost update anomalies do not take place in the READ COMMITTED isolation level. One can see that in Session A on Step 3, the update command runs successfully but with no effect taking place on the row that previously got deleted by Session B on Step 2.

3.3.6 Ghost update and REPEATABLE READ

SESSION A

SESSION B

```

MYSQL - TEST6 - # Anomaly|Ghost Update - Isolation|Repeatable Read
File Edit View Terminal Tabs Help

MariaDB [testdb]>
MariaDB [testdb]> -- S T E P 1 --
MariaDB [testdb]> -----
MariaDB [testdb]> SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> BEGIN;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> SELECT id,x FROM T ORDER BY id;
+----+-----+
| id | x   |
+----+-----+
| 1  | 1   |
| 2  | 2   |
| 3  | 3   |
+----+-----+
3 rows in set (0.000 sec)

MariaDB [testdb]> -- S T E P 3 --
MariaDB [testdb]> -----
MariaDB [testdb]> UPDATE T SET x=13 WHERE id=1;
Query OK, 0 rows affected (0.000 sec)
Rows matched: 0  Changed: 0  Warnings: 0

MariaDB [testdb]> SELECT id,x FROM T ORDER BY id;
+----+-----+
| id | x   |
+----+-----+
| 1  | 1   |
| 2  | 2   |
| 3  | 3   |
+----+-----+
3 rows in set (0.000 sec)

MariaDB [testdb]> COMMIT;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> SELECT id,x FROM T ORDER BY id;
+----+-----+
| id | x   |
+----+-----+
| 2  | 2   |
| 3  | 3   |
+----+-----+
2 rows in set (0.004 sec)

MariaDB [testdb]> COMMIT;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]>

MariaDB [testdb]>
MariaDB [testdb]> -- S T E P 2 --
MariaDB [testdb]> -----
MariaDB [testdb]> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> BEGIN;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> DELETE FROM T WHERE id=1;
Query OK, 1 row affected (0.000 sec)

MariaDB [testdb]> SELECT id,x FROM T ORDER BY id;
+----+-----+
| id | x   |
+----+-----+
| 2  | 2   |
| 3  | 3   |
+----+-----+
2 rows in set (0.000 sec)

MariaDB [testdb]> COMMIT;
Query OK, 0 rows affected (0.003 sec)

MariaDB [testdb]>

```

Figure 3.7: Ghost update in MySQL's REPEATABLE READ isolation level

The REPEATABLE READ isolation level also prevents ghost update phenomena. Interestingly enough, even though in Session A on Step 3 we can see that the row with $id = 1$ gets read, the `UPDATE` query doesn't have an effect on it.

3.3.7 Phantom update and READ COMMITTED

SESSION A

SESSION B

```

MYSQL - TEST7 - # Anomaly|Phantom Update - Isolation|Read Committed
File Edit View Terminal Tabs Help

MariaDB [testdb]>
MariaDB [testdb]> -- STEP 1 --
MariaDB [testdb]> -----
MariaDB [testdb]> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> BEGIN;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> SELECT * FROM T;
+-----+
| id | x |
+-----+
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
+-----+
3 rows in set (0.000 sec)

MariaDB [testdb]> -- STEP 3 --
MariaDB [testdb]> -----
MariaDB [testdb]> UPDATE T SET x=42 WHERE id=4;
Query OK, 1 row affected (0.000 sec)
Rows matched: 1 Changed: 1 Warnings: 0

MariaDB [testdb]> COMMIT;
Query OK, 0 rows affected (0.003 sec)

MariaDB [testdb]> -- STEP 4 --
MariaDB [testdb]> -----
MariaDB [testdb]> SELECT * FROM T ORDER BY id;
+-----+
| id | x |
+-----+
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
| 4 | 42 |
+-----+
4 rows in set (0.001 sec)

MariaDB [testdb]> COMMIT;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]>

MariaDB [testdb]>
MariaDB [testdb]> -- STEP 2 --
MariaDB [testdb]> -----
MariaDB [testdb]> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> BEGIN;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> INSERT INTO T VALUES (4,4);
Query OK, 1 row affected (0.000 sec)

MariaDB [testdb]> SELECT * FROM T ORDER BY id;
+-----+
| id | x |
+-----+
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
| 4 | 4 |
+-----+
4 rows in set (0.000 sec)

MariaDB [testdb]> COMMIT;
Query OK, 0 rows affected (0.003 sec)

MariaDB [testdb]>

```

Figure 3.8: Phantom update in MySQL's READ COMMITTED isolation level

Phantom update phenomena occur in the READ COMMITTED isolation level. Session A on Step 3, successfully updates a record that it had not read before. That record got committed by the concurrent Session B on Step 2.

3.3.8 Phantom update and REPEATABLE READ

SESSION A

SESSION B

```

MYSQL - TEST8 - # Anomaly|Phantom Update - Isolation|Repeatable Read
File Edit View Terminal Tabs Help

MariaDB [testdb]> -- S T E P 1 --
MariaDB [testdb]> -----
MariaDB [testdb]> SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> BEGIN;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> SELECT id,x FROM T ORDER BY id;
+----+-----+
| id | x   |
+----+-----+
| 1  | 1   |
| 2  | 2   |
| 3  | 3   |
+----+-----+
3 rows in set (0.000 sec)

MariaDB [testdb]> -- S T E P 3 --
MariaDB [testdb]> -----
MariaDB [testdb]> UPDATE T SET x=42 WHERE id=4;
Query OK, 1 row affected (0.000 sec)
Rows matched: 1  Changed: 1  Warnings: 0

MariaDB [testdb]> SELECT id,x FROM T ORDER BY id;
+----+-----+
| id | x   |
+----+-----+
| 1  | 1   |
| 2  | 2   |
| 3  | 3   |
| 4  | 42  |
+----+-----+
4 rows in set (0.000 sec)

MariaDB [testdb]> COMMIT;
Query OK, 0 rows affected (0.003 sec)

MariaDB [testdb]> -- S T E P 4 --
MariaDB [testdb]> -----
MariaDB [testdb]> SELECT * FROM T ORDER BY id;
+----+-----+
| id | x   |
+----+-----+
| 1  | 41  |
| 2  | 2   |
| 3  | 3   |
| 4  | 42  |
+----+-----+
4 rows in set (0.000 sec)

MariaDB [testdb]> COMMIT;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]>

MariaDB [testdb]> system clear
MariaDB [testdb]>
MariaDB [testdb]> -- S T E P 2 --
MariaDB [testdb]> -----
MariaDB [testdb]> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> BEGIN;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> INSERT INTO T VALUES (4,4);
Query OK, 1 row affected (0.000 sec)

MariaDB [testdb]> UPDATE T SET x=41 WHERE id=1;
Query OK, 1 row affected (0.000 sec)
Rows matched: 1  Changed: 1  Warnings: 0

MariaDB [testdb]> SELECT * FROM T ORDER BY id;
+----+-----+
| id | x   |
+----+-----+
| 1  | 41  |
| 2  | 2   |
| 3  | 3   |
| 4  | 4   |
+----+-----+
4 rows in set (0.000 sec)

MariaDB [testdb]> COMMIT;
Query OK, 0 rows affected (0.004 sec)

MariaDB [testdb]>

```

Figure 3.9: Phantom update in MySQL's REPEATABLE READ isolation level

Unlike other Database Management Systems' similar transaction isolation levels, MySQL's REPEATABLE READ allows phantom updates. As one can observe, the "consistent read" takes place only for the records that existed at the beginning of the transaction and not for the whole table. As a result, newly inserted records can be seen and acted upon.

In the example above, we see that the newly inserted record of Session B on Step 2 ($x = 4$) gets successfully updated by Session A's Step 3 even though it had not read this record in Step 1. Also, Session B's update on Step 2 ($x = 41$) is not seen by Session A on Step 3, but only after a commit takes place (Step 4).

3.3.9 Read skew and READ COMMITTED

Read skew anomaly is going to get examined using the following table which is a simulation of two bank accounts belonging to a user. The user is attempting to transfer money from one account to the other.

```
CREATE TABLE accounts (account_id INT NOT NULL PRIMARY KEY, balance INT);
INSERT INTO accounts VALUES (1, 500);
INSERT INTO accounts VALUES (2, 500);
```

SESSION A

SESSION B

```

MySQL - TEST9 - # Anomaly|Read Skew - Isolation|Read Committed

MariaDB [testdb]> -- STEP 1 --
MariaDB [testdb]> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> BEGIN;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> SELECT * FROM accounts;
+-----+
| account_id | balance |
+-----+
| 1 | 500 |
| 2 | 500 |
+-----+
2 rows in set (0.000 sec)

MariaDB [testdb]> -- STEP 3 --
MariaDB [testdb]> UPDATE accounts SET balance = balance - 100 WHERE account_id = 1;
Query OK, 1 row affected (0.003 sec)
Rows matched: 1 Changed: 1 Warnings: 0

MariaDB [testdb]> SELECT * FROM accounts;
+-----+
| account_id | balance |
+-----+
| 1 | 400 |
| 2 | 500 |
+-----+
2 rows in set (0.000 sec)

MariaDB [testdb]> -- STEP 5 --
MariaDB [testdb]> UPDATE accounts SET balance = balance + 100 WHERE account_id = 2;
Query OK, 1 row affected (0.004 sec)
Rows matched: 1 Changed: 1 Warnings: 0

MariaDB [testdb]> COMMIT;
Query OK, 0 rows affected (0.001 sec)

MariaDB [testdb]> SELECT * from accounts;
+-----+
| account_id | balance |
+-----+
| 1 | 400 |
| 2 | 600 |
+-----+
2 rows in set (0.001 sec)

MariaDB [testdb]> COMMIT;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]>

MariaDB [testdb]> system clear
MariaDB [testdb]> -- STEP 2 --
MariaDB [testdb]> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> BEGIN;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> SELECT * FROM accounts;
+-----+
| account_id | balance |
+-----+
| 1 | 500 |
| 2 | 500 |
+-----+
2 rows in set (0.000 sec)

MariaDB [testdb]> -- STEP 4 --
MariaDB [testdb]> SELECT balance from accounts WHERE account_id = 2;
+-----+
| balance |
+-----+
| 500 |
+-----+
1 row in set (0.000 sec)

MariaDB [testdb]> -- STEP 6 --
MariaDB [testdb]> SELECT balance from accounts WHERE account_id = 1;
+-----+
| balance |
+-----+
| 400 |
+-----+
1 row in set (0.001 sec)

MariaDB [testdb]> COMMIT;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]>

```

Figure 3.10: Read skew in MySQL's READ COMMITTED isolation level

In Steps 3 and 5 of Session A of Figure 3.10, an attempt is made to transfer **€100** from account 1 to account 2. In Step 4 of Session B of Figure 3.10, the current balance of account 2 is read before Step 5 (of Session A) adds **€100** to it. The value read equals **€500**. Next, Step 6 of Session B reads the current balance of account 1, following the subtraction of **€100**

in Step 3 of Session A. The value read is €400.

As a result, the transaction in Session B has read a combined value of accounts 1 and 2 that is equal to $500 + 400 = \text{€}900$ instead of the correct value which is **€1000**. This comprises the so called read skew problem. In essence, it is a violation of the data integrity constraint requiring the combined value of the two bank accounts to always remain equal to €1000. MySQL's READ COMMITTED isolation level is prone to read skew anomalies.

3.3.10 Read skew and REPEATABLE READ

SESSION A

SESSION B

```

MYSQ - TEST10 - # Anomaly|Read Skew - Isolation|Repeatable Read

File Edit View Terminal Tabs Help

MariaDB [testdb]> -- STEP 1 --
MariaDB [testdb]> SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
Query OK, 0 rows affected (0.003 sec)

MariaDB [testdb]> BEGIN;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> SELECT * FROM accounts;
+-----+
| account_id | balance |
+-----+
|          1 |      500 |
|          2 |      500 |
+-----+
2 rows in set (0.000 sec)

MariaDB [testdb]> -- STEP 3 --
MariaDB [testdb]> UPDATE accounts SET balance = balance - 100 WHERE account_id = 1;
Query OK, 1 row affected (0.000 sec)
Rows matched: 1 Changed: 1 Warnings: 0

MariaDB [testdb]> SELECT * FROM accounts;
+-----+
| account_id | balance |
+-----+
|          1 |      400 |
|          2 |      500 |
+-----+
2 rows in set (0.000 sec)

MariaDB [testdb]> -- STEP 5 --
MariaDB [testdb]> UPDATE accounts SET balance = balance + 100 WHERE account_id = 2;
Query OK, 1 row affected (0.000 sec)
Rows matched: 1 Changed: 1 Warnings: 0

MariaDB [testdb]> COMMIT;
Query OK, 0 rows affected (0.003 sec)

MariaDB [testdb]> SELECT * from accounts;
+-----+
| account_id | balance |
+-----+
|          1 |      400 |
|          2 |      600 |
+-----+
2 rows in set (0.000 sec)

MariaDB [testdb]> COMMIT;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]>

MariaDB [testdb]>

MariaDB [testdb]> -- STEP 2 --
MariaDB [testdb]> SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> BEGIN;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> SELECT * FROM accounts;
+-----+
| account_id | balance |
+-----+
|          1 |      500 |
|          2 |      500 |
+-----+
2 rows in set (0.000 sec)

MariaDB [testdb]> -- STEP 4 --
MariaDB [testdb]> SELECT balance from accounts WHERE account_id = 2;
+-----+
| balance |
+-----+
|      500 |
+-----+
1 row in set (0.000 sec)

MariaDB [testdb]> -- STEP 6 --
MariaDB [testdb]> SELECT balance from accounts WHERE account_id = 1;
+-----+
| balance |
+-----+
|      500 |
+-----+
1 row in set (0.000 sec)

MariaDB [testdb]> COMMIT;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]>

```

Figure 3.11: Read skew in MySQL's REPEATABLE READ isolation level

By definition, read skew is not expected to occur in the REPEATABLE READ isolation level. So, no matter *when* Session B reads each bank account, it will consistently read 500 for each one of them.

3.3.11 Write skew and REPEATABLE READ

To examine the write skew anomaly, the SQL code below is used.

The code creates a hospital table with doctors and their "on call" status.

Note: The hospital database model has a constraint requiring that **at least one** doctor must always be on call.

```
CREATE TABLE doctors (name VARCHAR2(255) NOT NULL PRIMARY KEY,
                      on_call VARCHAR(10), shift_id INT);
INSERT INTO doctors (name, on_call, shift_id)
VALUES ('Alice', 'true', 1234);
INSERT INTO doctors (name, on_call, shift_id)
VALUES ('Bob', 'true', 1234);
INSERT INTO doctors (name, on_call, shift_id)
VALUES ('Carol', 'false', 1234);
CREATE PROCEDURE currently_on_call(OUT doctors_on_call INT)
BEGIN
  SELECT count(*) INTO doctors_on_call
  FROM doctors WHERE on_call="true" AND shift_id=1234;
END
```

Figure 3.12 below illustrates a case where Alice and Bob apply for a leave simultaneously.

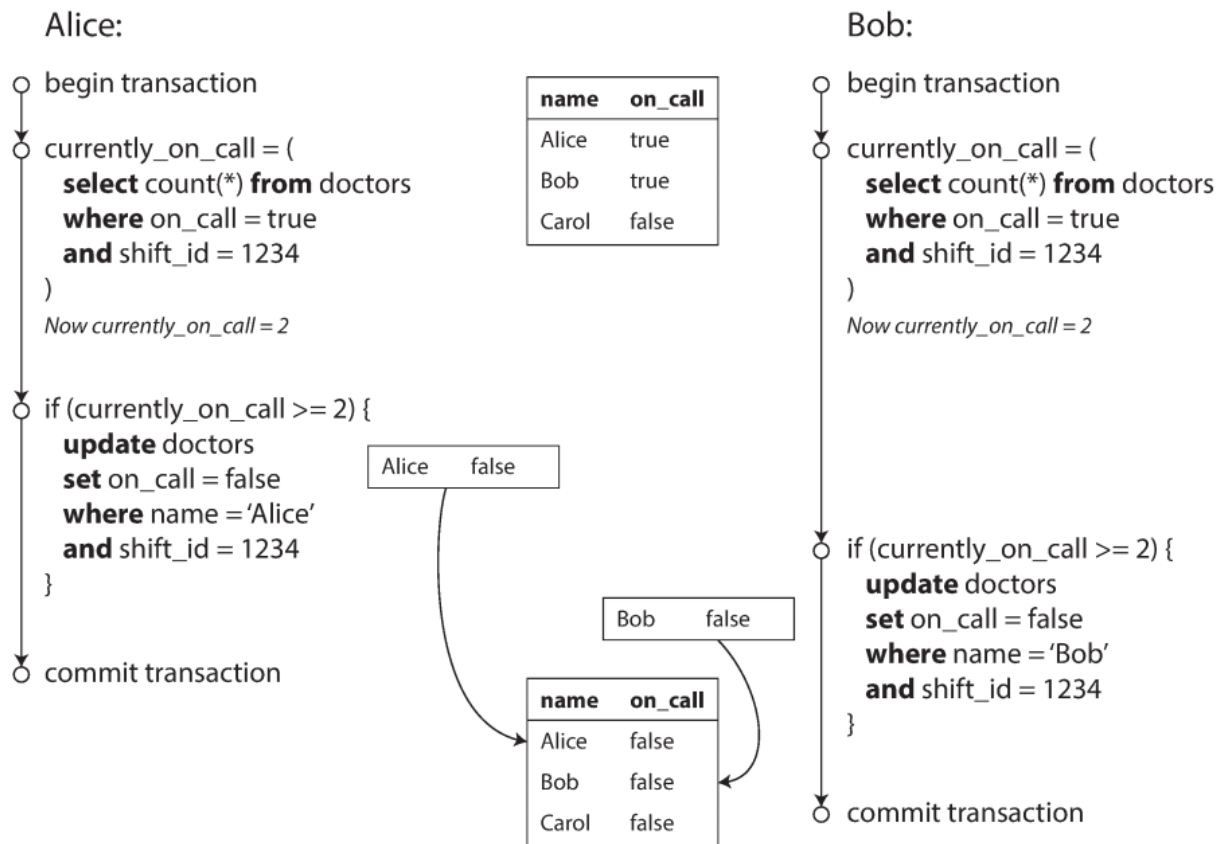


Figure 3.12: Example of write skew occurring (all false) [2]

SESSION A

SESSION B

```

MYSQ - TEST12 - # Anomaly|Write Skew - Isolation|Repeatable Read

File Edit View Terminal Tabs Help

MariaDB [testdb]> -- S T E P 1 --
MariaDB [testdb]> SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> BEGIN;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> SELECT * FROM doctors;
+-----+-----+-----+
| name | on_call | shift_id |
+-----+-----+-----+
| Alice | true   | 1234    |
| Bob   | true   | 1234    |
| Carol | false  | 1234    |
+-----+-----+-----+
3 rows in set (0.000 sec)

MariaDB [testdb]> CALL currently_on_call(@doctors_on_call);
Query OK, 1 row affected (0.000 sec)

MariaDB [testdb]> SELECT @doctors_on_call;
+-----+
| @doctors_on_call |
+-----+
| 2                |
+-----+
1 row in set (0.000 sec)

MariaDB [testdb]> -- S T E P 3 --
MariaDB [testdb]> UPDATE doctors SET on_call = IF(@doctors_on_call >= 2,
"false", "true") WHERE name = "Alice" and shift_id = 1234;
Query OK, 1 row affected (0.000 sec)
Rows matched: 1 Changed: 1 Warnings: 0

MariaDB [testdb]> SELECT * FROM doctors;
+-----+-----+-----+
| name | on_call | shift_id |
+-----+-----+-----+
| Alice | false   | 1234    |
| Bob   | true    | 1234    |
| Carol | false   | 1234    |
+-----+-----+-----+
3 rows in set (0.000 sec)

MariaDB [testdb]> COMMIT;
Query OK, 0 rows affected (0.003 sec)

MariaDB [testdb]>

MariaDB [testdb]> SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> BEGIN;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> SELECT * FROM doctors;
+-----+-----+-----+
| name | on_call | shift_id |
+-----+-----+-----+
| Alice | true    | 1234    |
| Bob   | true    | 1234    |
| Carol | false   | 1234    |
+-----+-----+-----+
3 rows in set (0.000 sec)

MariaDB [testdb]> -- S T E P 4 --
MariaDB [testdb]> CALL currently_on_call(@doctors_on_call);
Query OK, 1 row affected (0.001 sec)

MariaDB [testdb]> SELECT @doctors_on_call;
+-----+
| @doctors_on_call |
+-----+
| 2                |
+-----+
1 row in set (0.000 sec)

MariaDB [testdb]> UPDATE doctors SET on_call = IF(@doctors_on_call >= 2,
"false", "true") WHERE name = "Bob" and shift_id = 1234;
Query OK, 1 row affected (0.000 sec)
Rows matched: 1 Changed: 1 Warnings: 0

MariaDB [testdb]> SELECT * FROM doctors;
+-----+-----+-----+
| name | on_call | shift_id |
+-----+-----+-----+
| Alice | true    | 1234    |
| Bob   | false   | 1234    |
| Carol | false   | 1234    |
+-----+-----+-----+
3 rows in set (0.000 sec)

MariaDB [testdb]> COMMIT;
Query OK, 0 rows affected (0.003 sec)

MariaDB [testdb]> -- S T E P 5 --
MariaDB [testdb]> SELECT * FROM doctors;
+-----+-----+-----+
| name | on_call | shift_id |
+-----+-----+-----+
| Alice | false   | 1234    |
| Bob   | false   | 1234    |
| Carol | false   | 1234    |
+-----+-----+-----+
3 rows in set (0.000 sec)

MariaDB [testdb]> COMMIT;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]>

```

Figure 3.13: Write skew in MySQL's REPEATABLE READ isolation level

The REPEATABLE READ isolation level is known to not prevent the write skew problem. MySQL's REPEATABLE READ is no exception to the rule.

In Step 4 of Session B we note that the change at Step 3 of Session A, which changed Alice's `on_call` status to false, is not seen and as a result the count of `doctors_on_call` remains 2. So, Step 4 succeeds in updating Bob's `on_call` status to 'false'. After Session B commits at Step 4, we see that now all doctors are out of call. This violates the hospital's model constraint mentioned above.

3.3.12 Write skew and READ COMMITTED

SESSION A

SESSION B

```

MYSQ - TEST11 - # Anomaly|Write Skew - Isolation|Read Committed

File Edit View Terminal Tabs Help

MariaDB [testdb]> -- STEP 1 --
MariaDB [testdb]> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> BEGIN;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> SELECT * FROM doctors;
+-----+-----+-----+
| name | on_call | shift_id |
+-----+-----+-----+
| Alice | true    | 1234    |
| Bob   | true    | 1234    |
| Carol | false   | 1234    |
+-----+-----+-----+
3 rows in set (0.000 sec)

MariaDB [testdb]> CALL currently_on_call(@doctors_on_call);
Query OK, 1 row affected (0.001 sec)

MariaDB [testdb]> SELECT @doctors_on_call;
+-----+
| @doctors_on_call |
+-----+
| 2                |
+-----+
1 row in set (0.000 sec)

MariaDB [testdb]> -- STEP 3 --
MariaDB [testdb]> UPDATE doctors SET on_call = IF(@doctors_on_call >= 2,
"false", "true") WHERE name = "Alice" and shift_id = 1234;
Query OK, 1 row affected (0.000 sec)
Rows matched: 1 Changed: 1 Warnings: 0

MariaDB [testdb]> SELECT * FROM doctors;
+-----+-----+-----+
| name | on_call | shift_id |
+-----+-----+-----+
| Alice | false   | 1234    |
| Bob   | true    | 1234    |
| Carol | false   | 1234    |
+-----+-----+-----+
3 rows in set (0.000 sec)

MariaDB [testdb]> COMMIT;
Query OK, 0 rows affected (0.002 sec)

MariaDB [testdb]>

11/77 MariaDB [testdb]>
MariaDB [testdb]> -- STEP 2 --
MariaDB [testdb]> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> BEGIN;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> SELECT * FROM doctors;
+-----+-----+-----+
| name | on_call | shift_id |
+-----+-----+-----+
| Alice | true    | 1234    |
| Bob   | true    | 1234    |
| Carol | false   | 1234    |
+-----+-----+-----+
3 rows in set (0.000 sec)

MariaDB [testdb]> -- STEP 4 --
MariaDB [testdb]> CALL currently_on_call(@doctors_on_call);
Query OK, 1 row affected (0.001 sec)

MariaDB [testdb]> SELECT @doctors_on_call;
+-----+
| @doctors_on_call |
+-----+
| 1                |
+-----+
1 row in set (0.000 sec)

MariaDB [testdb]> UPDATE doctors SET on_call = IF(@doctors_on_call >= 2, "
false", "true") WHERE name = "Bob" and shift_id = 1234;
Query OK, 0 rows affected (0.000 sec)
Rows matched: 1 Changed: 0 Warnings: 0

MariaDB [testdb]> COMMIT;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> -- STEP 5 --
MariaDB [testdb]> SELECT * FROM doctors;
+-----+-----+-----+
| name | on_call | shift_id |
+-----+-----+-----+
| Alice | false   | 1234    |
| Bob   | true    | 1234    |
| Carol | false   | 1234    |
+-----+-----+-----+
3 rows in set (0.000 sec)

MariaDB [testdb]> COMMIT;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]>

```

Figure 3.14: Write skew in MySQL's READ COMMITTED isolation level

As one can expect, due to its nature, the READ COMMITTED isolation levels are not vulnerable to write skew phenomena.

So, we see in Step 4, that the second transaction correctly sees 1 available doctor on call and as a result Bob is not allowed to take his absence leave, because the hospital must have at least 1 doctor on call.

3.4 Comparison of test results

This table sums up the findings of how MySQL isolation levels behave when encountering the tested transaction anomalies.

Table 3.1: MySQL isolation levels and transaction anomalies

	Read Committed	Repeatable Read
NON-REPEATABLE READ	Possible	Not Possible
LOST UPDATE	Possible	Possible
GHOST UPDATE	Not Possible ¹	Not Possible ¹
PHANTOM UPDATE	Possible	Possible ²
READ SKEW	Possible	Not Possible
WRITE SKEW	Not Possible	Possible

¹ Exception does not occur.

² Without phantom read.

Chapter 4

MVCC in PostgreSQL

4.1 PostgreSQL Transaction Isolation Levels

There are four transaction isolation levels defined in the ANSI SQL '92 Standard:

- READ UNCOMMITTED
- READ COMMITTED
- REPEATABLE READ
- SERIALIZABLE

Even though all of the four can be requested in PostgreSQL, only the last three are implemented internally. *PostgreSQL's READ UNCOMMITTED mode behaves like READ COMMITTED. This is because it is the only sensible way to map the standard isolation levels to PostgreSQL's multiversion concurrency control architecture. [20]*

4.1.1 READ COMMITTED

This is PostgreSQL's default transaction isolation level. **SELECT** queries of a transaction using this isolation level will see data that got committed exactly before the query took place. Uncommitted changes, or changes that got committed after the query executed, will not be seen. Uncommitted changes within the same transaction will be seen.

Transaction anomalies that are bound to happen when using this isolation level are non-repeatable reads, lost updates, phantom reads, phantom updates as well as skewed reads.

The partial transaction isolation provided by the **READ COMMITTED** isolation level is sufficient for many applications because it is fast and simple to use. The simplicity lies in the fact that a transaction abort is possible only in the case of a failure, but not as a means to prevent inconsistency [21], because a serialization error cannot occur. However, it is not suitable for all cases. Applications that do complex queries and updates might require a more rigorously consistent view of the database than the **READ COMMITTED** isolation level provides [20].

4.1.2 REPEATABLE READ

In this isolation level, **SELECT** queries see data committed before the start of the transaction but not committed before each statement in the current transaction, as is the case in **READ COMMITTED** isolation level. Uncommitted changes or committed changes by other concurrent transactions are never seen. This provides the transaction with a more consistent view of the database.

If a transaction, running with the **REPEATABLE READ** isolation level, tries to update or delete a row that got changed and committed by another concurrent transaction, the former will be rolled back with the message **"ERROR: could not serialize access due to concurrent update"**. Therefore, developers that are using this isolation level, should bear this error in mind and handle it appropriately (e.g., by retrying the transaction).

Although most transaction anomalies are prevented by this isolation level, write skew anomalies do occur. This makes it a convenient isolation level to be used in applications that use mostly read-only transactions, such as report building applications.

4.1.3 SERIALIZABLE

This isolation level makes concurrent transactions behave like if they would run serially, one after the other. Internally, there aren't any significant differences between **SERIALIZABLE** and **REPEATABLE READ** except that **SERIALIZABLE** *monitors for conditions which could make execution of a concurrent set of serializable transactions behave in a manner inconsistent with all possible serial executions of those transactions. This monitoring does not introduce any blocking beyond that present in REPEATABLE READ, but there is some overhead to the monitoring, and detection of the conditions which could cause a serialization failure.* [20]

The **SERIALIZABLE** isolation level, makes a developer not worry about inconsistencies but only handle serialization errors by repeating the transactions. Applications using this isolation level will have significantly reduced system throughput due to the additional overhead.

4.2 PostgreSQL and MVCC

Unlike Oracle, MySQL and SQL Server, which use a mechanism of storing each modification in a separate data structure (e.g., Undo Segments), PostgreSQL's approach could be considered more elegant. PostgreSQL stores multiple rows in the table's data structure itself and limits their visibility with the use **xmin** and **xmax** system columns [4].

When a transaction starts, the **XID** number (a transaction ID) gets incremented and assigned to the current transaction.

In the case of an **INSERT**, the **XID** gets stored in the row as the value of **xmin**. Every row that has been committed and has the **xmin** value less than the current transaction's **XID** is visible to the transaction. This means that you can start a transaction and insert a row, and until that transaction commits, that row will not be visible to other transactions. Once it commits, other transactions will be able to view the new row because they will satisfy the **xmin <= XID and xmax >= XID** condition. The underlying mechanism works the same for **DELETES** and **UPDATES**, only in these cases it stores the **xmax** value (instead of **xmin**) on each row in order to determine visibility.

In the case of a DELETE, the XID value is stored in the `xmax` column and the row is marked for deletion, without actually being deleted. This is because, an already open transaction must be able to see a previous version of that row. Actual data deletion occurs when the mechanism called `vacuum` takes action. *This process looks up tuples that are no longer accessible to any transaction and deletes them. Without vacuuming the database would only grow in size, and even worse, it would eventually hit a transaction ID wraparound that can have catastrophic consequences [4].*

An UPDATE is essentially a deletion of the tuple and then an insertion. In Figure 4.1 one can see that each process (A and B) sees a different tuple. Process B sees the old tuple with its old values but the `xmax` value has been updated with the XID value of the transaction A that updated (deleted and inserted) the tuple.

```
process A: BEGIN;
process A: SELECT txid_current();

txid_current
-----
          23

process A: UPDATE users SET name = 'Marko' WHERE id = 1;
-- Traditionally, this would lock the record for reading.

process A: SELECT *, xmin, xmax FROM users;
 id | name  | xmin | xmax
----+-----+-----+-----
  1 | Marko |   23 |    0
(1 rows)

-- Process A creates a new tuple and sets the xmin to its transaction id

process B: SELECT *, xmin, xmax FROM users;
 id | name  | xmin | xmax
----+-----+-----+-----
  1 | John  |   17 |   23
(1 rows)

-- Process B still see the old tuple. Notice that this tuple has
-- its end of life set to Process A's transaction id.
```

Figure 4.1: PostgreSQL's UPDATE under the hood [4]

4.3 Examples of isolation levels

The following scenarios are being tested in two, side-by-side active concurrent transactions. The table that is being used got created with this SQL code:

```
CREATE TABLE T (id INT NOT NULL PRIMARY KEY, x INT);
INSERT INTO T VALUES (1, 1);
INSERT INTO T VALUES (2, 2);
INSERT INTO T VALUES (3, 3);
```

4.3.1 Non-repeatable read and READ COMMITTED

SESSION A

SESSION B

```

POSTGRES - TEST1 - # Anomaly|Non-Repeatable Read - Isolation|Read Committed

File Edit View Terminal Tabs Help

test=#
test=# -- STEP 1 --
test=# -----
test=# BEGIN TRANSACTION;
BEGIN
test=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SET
test=# SELECT * FROM T WHERE id=1;
id | x
---+---
 1 | 1
(1 row)

test=# -- STEP 3 --
test=# -----
test=# SELECT * FROM T ORDER BY id;
id | x
---+---
 1 | 1
 2 | 2
 3 | 3
(3 rows)

test=# -- STEP 5 --
test=# -----
test=# SELECT * FROM T ORDER BY id;
id | x
---+---
 1 | 12
 2 | 2
 3 | 3
(3 rows)

test=# COMMIT;
COMMIT
test=#

test=#
test=# -- STEP 2 --
test=# -----
test=# BEGIN TRANSACTION;
BEGIN
test=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SET
test=# UPDATE T SET x=12 WHERE id=1;
UPDATE 1
test=# SELECT * FROM T WHERE id=1;
id | x
---+---
 1 | 12
(1 row)

test=# -- STEP 4 --
test=# -----
test=# COMMIT;
COMMIT
test=#

```

Figure 4.2: Non-repeatable read in Postgres’s READ COMMITTED isolation level

This example demonstrates that PostgreSQL’s READ COMMITTED isolation level does not provide a repeatable read. If that was the case, Session A on Step 5 would not see the value of 12 (that Session B committed at Step 4) but the value of 1, as it did at the beginning of the transaction. Also, Step 3 (Session A) demonstrates that a Dirty Read is not possible, as the not-yet-committed value of 12 is not seen.

4.3.2 Non-repeatable read and REPEATABLE READ

SESSION A

SESSION B

```

POSTGRES - TEST2 - # Anomaly|Non-Repeatable Read - Isolation|Repeatable Read

File Edit View Terminal Tabs Help

test=#
test=# -- S T E P 1 --
test=# -----
test=# BEGIN TRANSACTION;
test=# BEGIN
test=# SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
test=# SET
test=# SELECT * FROM T WHERE id=1;
test=# id | x
test=# -----
test=# 1 | 1
test=# (1 row)
test=#
test=# -- S T E P 3 --
test=# -----
test=# SELECT * FROM T WHERE id=1;
test=# id | x
test=# -----
test=# 1 | 1
test=# (1 row)
test=# COMMIT;
test=# COMMIT
test=# SELECT * FROM T WHERE id=1;
test=# id | x
test=# -----
test=# 1 | 12
test=# (1 row)
test=# COMMIT;
test=# COMMIT
test=#

test=#
test=# -- S T E P 2 --
test=# -----
test=# BEGIN TRANSACTION;
test=# BEGIN
test=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
test=# SET
test=# UPDATE T SET x=12 WHERE id=1;
test=# UPDATE 1
test=# SELECT * FROM T WHERE id=1;
test=# id | x
test=# -----
test=# 1 | 12
test=# (1 row)
test=# COMMIT;
test=# COMMIT
test=#

```

Figure 4.3: Non-repeatable read in Postgres's REPEATABLE READ isolation level

The repeatable read, of course, is provided when the REPEATABLE READ isolation level is selected for the transaction. And this is the reason why the committed change of Session B at Step 2 is only seen after Session A commits (Step 3).

4.3.3 Non-repeatable read and SERIALIZABLE

SESSION A

SESSION B

```

POSTGRES - TEST3 - # Anomaly|Non-Repeatable Read - Isolation|Serializable

File Edit View Terminal Tabs Help

test=#
test=# -- S T E P 1 --
test=# -----
test=# BEGIN TRANSACTION;
test=# BEGIN
test=# SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
test=# SET
test=# SELECT * FROM T WHERE id=1;
test=# id | x
test=# -----
test=# 1 | 1
test=# (1 row)
test=#
test=# -- S T E P 3 --
test=# -----
test=# SELECT * FROM T WHERE id=1;
test=# id | x
test=# -----
test=# 1 | 1
test=# (1 row)
test=# COMMIT;
test=# COMMIT
test=#

test=#
test=# -- S T E P 2 --
test=# -----
test=# BEGIN TRANSACTION;
test=# BEGIN
test=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
test=# SET
test=# UPDATE T SET x=12 WHERE id=1;
test=# UPDATE 1
test=# SELECT * FROM T WHERE id=1;
test=# id | x
test=# -----
test=# 1 | 12
test=# (1 row)
test=# COMMIT;
test=# COMMIT
test=#

```

Figure 4.4: Non-repeatable read in Postgres's SERIALIZABLE isolation level

As expected for this anomaly, the SERIALIZABLE isolation level behaves exactly the same as the REPEATABLE READ.

4.3.4 Lost update and READ COMMITTED

SESSION A

SESSION B

```

POSTGRES - TEST4 - # Anomaly|Lost update - Isolation|Read Committed

File Edit View Terminal Tabs Help

test=# -- STEP 1 --
test=# -----
test=# BEGIN TRANSACTION;
test=# BEGIN
test=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
test=# SET
test=# SELECT * FROM T WHERE id=1;
id | x
---+---
 1 | 1
(1 row)

test=# -- STEP 3 --
test=# -----
test=# UPDATE T SET x=13 WHERE id=1;
UPDATE 1
test=# SELECT * FROM T WHERE id=1;
id | x
---+---
 1 | 13
(1 row)

test=# COMMIT;
COMMIT
test=# SELECT * FROM T WHERE id=1;
id | x
---+---
 1 | 13
(1 row)

test=# COMMIT;
COMMIT
test=#

test=# -- STEP 2 --
test=# -----
test=# BEGIN TRANSACTION;
test=# BEGIN
test=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
test=# SET
test=# UPDATE T SET x=12 WHERE id=1;
UPDATE 1
test=# SELECT * FROM T WHERE id=1;
id | x
---+---
 1 | 12
(1 row)

test=# COMMIT;
COMMIT
test=#

```

Figure 4.5: Lost update in Postgres's READ COMMITTED isolation level

Here one can see the classic behaviour of the READ COMMITTED isolation level that overwrites a value (Step 3 of Session A) that got also updated (and committed) in Session B on Step 2 without throwing an exception or a warning.

4.3.5 Lost update and REPEATABLE READ

SESSION A

SESSION B

```

POSTGRES - TEST5 - # Anomaly|Lost update - Isolation|Repeatable Read

File Edit View Terminal Tabs Help

test=# -- S T E P 1 --
test=# -----
test=# BEGIN TRANSACTION;
BEGIN
test=# SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SET
test=# SELECT * FROM T WHERE id=1;
 id | x
-----+-----
  1 | 1
(1 row)

test=# -- S T E P 3 --
test=# -----
test=# UPDATE T SET x=13 WHERE id=1;
ERROR:  could not serialize access due to concurrent update
test=# SELECT * FROM T WHERE id=1;
ERROR:  current transaction is aborted, commands ignored until end of transaction block
test=# COMMIT;
ROLLBACK
test=# SELECT * FROM T WHERE id=1;
 id | x
-----+-----
  1 | 12
(1 row)

test=# COMMIT;
COMMIT
test=#

test=# -- S T E P 2 --
test=# -----
test=# BEGIN TRANSACTION;
BEGIN
test=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SET
test=# UPDATE T SET x=12 WHERE id=1;
UPDATE 1
test=# SELECT * FROM T WHERE id=1;
 id | x
-----+-----
  1 | 12
(1 row)

test=# COMMIT;
COMMIT
test=#

```

Figure 4.6: Lost update in Postgres's REPEATABLE READ isolation level

The REPEATABLE READ isolation level protects against lost update phenomena. This can be seen in Session A, where Step 3 tries to update a tuple that got updated and committed by Session B on Step 2. An exception **ERROR: could not serialize access due to concurrent update** is thrown and the transaction automatically rolls back.

4.3.6 Lost update and SERIALIZABLE

SESSION A

SESSION B

```

POSTGRES - TEST6 - # Anomaly|Lost update - Isolation|Serializable

File Edit View Terminal Tabs Help

test=# -- S T E P 1 --
test=# -----
test=# BEGIN TRANSACTION;
test=# SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
test=# SET
test=# SELECT * FROM T WHERE id=1;
test=#      id | x
test=# -----
test=#      1 | 1
test=# (1 row)

test=# -- S T E P 3 --
test=# -----
test=# UPDATE T SET x=13 WHERE id=1;
test=# ERROR:  could not serialize access due to concurrent update
test=# SELECT * FROM T WHERE id=1;
test=# ERROR:  current transaction is aborted, commands ignored until end of tran
test=# saction block
test=# COMMIT;
test=# ROLLBACK
test=# SELECT * FROM T WHERE id=1;
test=#      id | x
test=# -----
test=#      1 | 12
test=# (1 row)

test=# COMMIT;
test=# COMMIT
test=#

test=# -- S T E P 2 --
test=# -----
test=# BEGIN TRANSACTION;
test=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
test=# SET
test=# UPDATE T SET x=12 WHERE id=1;
test=# UPDATE 1
test=# SELECT * FROM T WHERE id=1;
test=#      id | x
test=# -----
test=#      1 | 12
test=# (1 row)

test=# COMMIT;
test=# COMMIT
test=#

```

Figure 4.7: Lost update in Postgres's SERIALIZABLE isolation level

As we can see, SERIALIZABLE isolation level provides the exact same behaviour as that of the REPEATABLE READ isolation level.

4.3.7 Ghost update and READ COMMITTED

SESSION A

SESSION B

```

POSTGRES - TEST7 - # Anomaly|Ghost update - Isolation|Read Committed

File Edit View Terminal Tabs Help

test=#
test=# -- STEP 1 --
test=# -----
test=# BEGIN TRANSACTION;
BEGIN
test=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SET
test=# SELECT * FROM T ORDER BY id;
 id | x
-----+-----
  1 | 1
  2 | 2
  3 | 3
(3 rows)

test=# -- STEP 3 --
test=# -----
test=# UPDATE T SET x=12 WHERE id=1;
UPDATE 0
test=# COMMIT;
COMMIT
test=# SELECT * FROM T ORDER BY id;
 id | x
-----+-----
  2 | 2
  3 | 3
(2 rows)

test=# COMMIT;
COMMIT
test=#

test=#
test=# -- STEP 2 --
test=# -----
test=# BEGIN TRANSACTION;
BEGIN
test=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SET
test=# DELETE FROM T WHERE id=1;
DELETE 1
test=# SELECT * FROM T ORDER BY id;
 id | x
-----+-----
  2 | 2
  3 | 3
(2 rows)

test=# COMMIT;
COMMIT
test=#

```

Figure 4.8: Ghost update in Postgres's READ COMMITTED isolation level

Ghost update anomalies are not expected to occur in the READ COMMITTED isolation level. We can see that, in Session A, when Step 3 tries to update a row which got deleted by Session B on Step 2, the command successfully executes, but with no effect (UPDATE 0) and without throwing an exception or a warning.

4.3.8 Ghost update and REPEATABLE READ

SESSION A

SESSION B

```

POSTGRES - TEST8 - # Anomaly|Ghost update - Isolation|Repeatable Read

File Edit View Terminal Tabs Help

test=#
test=# -- STEP 1 --
test=# -----
test=# BEGIN TRANSACTION;
test=# BEGIN
test=# SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
test=# SET
test=# SELECT * FROM T ORDER BY id;
id | x
----+--
 1 | 1
 2 | 2
 3 | 3
(3 rows)

test=# -- STEP 3 --
test=# -----
test=# UPDATE T SET x=12 WHERE id=1;
ERROR: could not serialize access due to concurrent update
test=# COMMIT;
ROLLBACK
test=# SELECT * FROM T ORDER BY id;
id | x
----+--
 2 | 2
 3 | 3
(2 rows)

test=# COMMIT;
COMMIT
test=#

test=#
test=# -- STEP 2 --
test=# -----
test=# BEGIN TRANSACTION;
test=# BEGIN
test=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
test=# SET
test=# DELETE FROM T WHERE id=1;
DELETE 1
test=# SELECT * FROM T ORDER BY id;
id | x
----+--
 2 | 2
 3 | 3
(2 rows)

test=# COMMIT;
COMMIT
test=#

```

Figure 4.9: Ghost update in Postgres's REPEATABLE READ isolation level

The REPEATABLE READ isolation level is prone to ghost update anomalies, but PostgreSQL's REPEATABLE READ prevents them from happening by stopping the execution with a message `ERROR: could not serialize access due to concurrent update` (Session A Step 3) and rolling back the transaction automatically.

4.3.9 Ghost update and SERIALIZABLE

SESSION A

SESSION B

```

POSTGRES - TEST9 - # Anomaly|Ghost update - Isolation|Serializable

test=# -- STEP 1 --
test=# -----
test=# BEGIN TRANSACTION;
test=# BEGIN
test=# SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
test=# SET
test=# SELECT * FROM T ORDER BY id;
id | x
----+--
1  | 1
2  | 2
3  | 3
(3 rows)

test=# -- STEP 3 --
test=# -----
test=# UPDATE T SET x=12 WHERE id=1;
ERROR:  could not serialize access due to concurrent update
test=# COMMIT;
ROLLBACK
test=# SELECT * FROM T ORDER BY id;
id | x
----+--
2  | 2
3  | 3
(2 rows)

test=# COMMIT;
test=#

```

```

test=# -- STEP 2 --
test=# -----
test=# BEGIN TRANSACTION;
test=# BEGIN
test=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
test=# SET
test=# DELETE FROM T WHERE id=1;
DELETE 1
test=# SELECT * FROM T ORDER BY id;
id | x
----+--
2  | 2
3  | 3
(2 rows)

test=# COMMIT;
test=# COMMIT
test=#

```

Figure 4.10: Ghost update in Postgres's SERIALIZABLE isolation level

Once again, the SERIALIZABLE isolation level handles the same anomaly like the REPEATABLE READ isolation level does.

4.3.10 Phantom update and READ COMMITTED

SESSION A

SESSION B

```

POSTGRES - TEST10 - # Anomaly|Phantom update - Isolation|Read Committed

File Edit View Terminal Tabs Help

test=#
test=# -- S T E P 1 --
test=# -----
test=# BEGIN TRANSACTION;
BEGIN
test=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SET
test=# SELECT * FROM T ORDER BY id;
 id | x
-----+-----
  1 | 1
  2 | 2
  3 | 3
(3 rows)

test=# -- S T E P 3 --
test=# -----
test=# UPDATE T SET x=42 WHERE id=4;
UPDATE 1
test=# SELECT id,x FROM T ORDER BY id;
 id | x
-----+-----
  1 | 1
  2 | 2
  3 | 3
  4 | 42
(4 rows)

test=# COMMIT;
COMMIT
test=# SELECT id,x FROM T ORDER BY id;
 id | x
-----+-----
  1 | 1
  2 | 2
  3 | 3
  4 | 42
(4 rows)

test=# COMMIT;
COMMIT
test=#

test=#
test=# -- S T E P 2 --
test=# -----
test=# BEGIN TRANSACTION;
BEGIN
test=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SET
test=# INSERT INTO T VALUES (4,4);
INSERT 0 1
test=# SELECT * FROM T ORDER BY id;
 id | x
-----+-----
  1 | 1
  2 | 2
  3 | 3
  4 | 4
(4 rows)

test=# COMMIT;
COMMIT
test=#

```

Figure 4.11: Phantom update in Postgres's READ COMMITTED isolation level

Session A in Step 3 will try to update a row that has not been read in Step 1. The update will be successful, since this row got inserted (and committed) by the concurrent Session B. This is the case of a phantom update that the READ COMMITTED isolation level did not prevent.

4.3.11 Phantom update and REPEATABLE READ

SESSION A

SESSION B

```

POSTGRES - TEST11 - # Anomaly|Phantom update - Isolation|Repeatable Read

File Edit View Terminal Tabs Help

test=#
test=# -- S T E P 1 --
test=# -----
test=# BEGIN TRANSACTION;
BEGIN
test=# SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SET
test=# SELECT * FROM T ORDER BY id;
id | x
-----+-----
1  | 1
2  | 2
3  | 3
(3 rows)

test=# -- S T E P 3 --
test=# -----
test=# UPDATE T SET x=42 WHERE id=4;
UPDATE 0
test=# SELECT id,x FROM T ORDER BY id;
id | x
-----+-----
1  | 1
2  | 2
3  | 3
(3 rows)

test=# COMMIT;
COMMIT
test=# SELECT id,x FROM T ORDER BY id;
id | x
-----+-----
1  | 1
2  | 2
3  | 3
4  | 4
(4 rows)

test=# COMMIT;
COMMIT
test=#

test=#
test=# -- S T E P 2 --
test=# -----
test=# BEGIN TRANSACTION;
BEGIN
test=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SET
test=# INSERT INTO T VALUES (4,4);
INSERT 0 1
test=# SELECT * FROM T ORDER BY id;
id | x
-----+-----
1  | 1
2  | 2
3  | 3
4  | 4
(4 rows)

test=# COMMIT;
COMMIT
test=#

```

Figure 4.12: Phantom update in Postgres's READ COMMITTED isolation level

In the REPEATABLE READ isolation level, Step 3 will fail to update the row that got inserted and committed by the second concurrent transaction. The phantom update anomaly got prevented without an exception or a warning.

4.3.12 Phantom update and SERIALIZABLE

SESSION A

SESSION B

```

POSTGRES - TEST12 - # Anomaly|Phantom update - Isolation|Serializable

File Edit View Terminal Tabs Help

test=#
test=# -- S T E P 1 --
test=# -----
test=# BEGIN TRANSACTION;
test=# BEGIN
test=# SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
test=# SET
test=# SELECT * FROM T ORDER BY id;
id | x
---+---
 1 | 1
 2 | 2
 3 | 3
(3 rows)

test=# -- S T E P 3 --
test=# -----
test=# UPDATE T SET x=42 WHERE id=4;
UPDATE 0
test=# SELECT id,x FROM T ORDER BY id;
id | x
---+---
 1 | 1
 2 | 2
 3 | 3
(3 rows)

test=# COMMIT;
COMMIT
test=# SELECT id,x FROM T ORDER BY id;
id | x
---+---
 1 | 1
 2 | 2
 3 | 3
 4 | 4
(4 rows)

test=# COMMIT;
COMMIT
test=#

test=#
test=# -- S T E P 2 --
test=# -----
test=# BEGIN TRANSACTION;
test=# BEGIN
test=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
test=# SET
test=# INSERT INTO T VALUES (4,4);
INSERT 0 1
test=# SELECT * FROM T ORDER BY id;
id | x
---+---
 1 | 1
 2 | 2
 3 | 3
 4 | 4
(4 rows)

test=# COMMIT;
COMMIT
test=#

```

Figure 4.13: Phantom update in Postgres's SERIALIZABLE isolation level

Similar to the REPEATABLE READ, SERIALIZABLE isolation level will also prevent a phantom update anomaly without an exception or a warning.

4.3.13 Read skew and READ COMMITTED

Read Skew anomaly is going to get examined using the following table which is a simulation of two bank accounts belonging to a user. The user is attempting to transfer money from one account to the other.

```
CREATE TABLE accounts (account_id INT NOT NULL PRIMARY KEY, balance INT);
INSERT INTO accounts VALUES (1, 500);
INSERT INTO accounts VALUES (2, 500);
```

SESSION A

SESSION B

```

test=# ---- S T E P 1 ----
test=# BEGIN TRANSACTION;
BEGIN
test=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SET
test=# SELECT * FROM accounts;
 account_id | balance 
-----+-----
          1 |      500
          2 |      500
(2 rows)

test=# ---- S T E P 3 ----
test=# UPDATE accounts SET balance = balance - 100 WHERE account_id = 1;
UPDATE 1
test=# SELECT * FROM accounts;
 account_id | balance 
-----+-----
          2 |      500
          1 |      400
(2 rows)

test=# ---- S T E P 5 ----
test=# UPDATE accounts SET balance = balance + 100 WHERE account_id = 2;
UPDATE 1
test=# COMMIT;
COMMIT
test=# SELECT * from accounts;
 account_id | balance 
-----+-----
          1 |      400
          2 |      600
(2 rows)

test=# COMMIT;
COMMIT
test=#

test=# ---- S T E P 2 ----
test=# BEGIN TRANSACTION;
BEGIN
test=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SET
test=# SELECT * FROM accounts;
 account_id | balance 
-----+-----
          1 |      500
          2 |      500
(2 rows)

test=# ---- S T E P 4 ----
test=# SELECT balance from accounts WHERE account_id = 2;
 balance 
-----
      500
(1 row)

test=# ---- S T E P 6 ----
test=# SELECT balance from accounts WHERE account_id = 1;
 balance 
-----
      400
(1 row)

test=# COMMIT;
COMMIT
test=#

```

Figure 4.14: Read skew in PostgreSQL's READ COMMITTED isolation level

In Steps 3 and 5 of the left session (Session A) of Figure 4.14, an attempt is made to transfer **€100** from account 1 to account 2. In Step 4 of the session in the right (Session B) of Figure 4.14, the current balance of account 2 is read before Step 5 (of Session A) adds **€100** to it. The value read equals **€500**. Next, Step 6 of Session B reads the current balance of account 1, following the subtraction of **€100** in Step 3 of Session A. The value read is **€400**. As a result, the transaction in Session B has read a combined value of accounts 1 and 2 that is equal to $500 + 400 = \text{€}900$ instead of the correct value which is **€1000**. This comprises the so called read skew problem. In essence, it is a violation of the data integrity constraint requiring the combined value of the two bank accounts to always remain equal to **€1000**. PostgreSQL's READ COMMITTED isolation level is prone to read skew anomalies.

4.3.14 Read skew and REPEATABLE READ

SESSION A

SESSION B

```

POSTGRES - TEST14 - # Anomaly|Read Skew - Isolation|Repeatable Read

File Edit View Terminal Tabs Help

test=# ---- S T E P 1 ----
test=# BEGIN TRANSACTION;
BEGIN
test=# SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SET
test=# SELECT * FROM accounts;
 account_id | balance 
-----
1 | 500
2 | 500
(2 rows)

test=# ---- S T E P 3 ----
test=# UPDATE accounts SET balance = balance - 100 WHERE account_id = 1;
UPDATE 1
test=# SELECT * FROM accounts;
 account_id | balance 
-----
2 | 500
1 | 400
(2 rows)

test=# ---- S T E P 5 ----
test=# UPDATE accounts SET balance = balance + 100 WHERE account_id = 2;
UPDATE 1
test=# COMMIT;
COMMIT
test=# SELECT * from accounts;
 account_id | balance 
-----
1 | 400
2 | 600
(2 rows)

test=# COMMIT;
COMMIT
test=#

test=# ---- S T E P 2 ----
test=# BEGIN TRANSACTION;
BEGIN
test=# SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SET
test=# SELECT * FROM accounts;
 account_id | balance 
-----
1 | 500
2 | 500
(2 rows)

test=# ---- S T E P 4 ----
test=# SELECT balance from accounts WHERE account_id = 2;
 balance 
-----
500
(1 row)

test=# ---- S T E P 6 ----
test=# SELECT balance from accounts WHERE account_id = 1;
 balance 
-----
500
(1 row)

test=# COMMIT;
COMMIT
test=#

```

Figure 4.15: Read skew in Postgres's REPEATABLE READ isolation level

Expectedly, the REPEATABLE READ isolation level will consistently read the same values in the beginning as well as in the end of the transaction. As a result, a read skew anomaly will always be prevented when using this isolation level.

4.3.15 Read skew and SERIALIZABLE

SESSION A

SESSION B

```

POSTGRES - TEST15 - # Anomaly|Read Skew - Isolation|Serializable

File Edit View Terminal Tabs Help

test=# ---- S T E P 1 ----
test=# BEGIN TRANSACTION;
BEGIN
test=# SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SET
test=# SELECT * FROM accounts;
 account_id | balance 
-----+-----
          1 |      500
          2 |      500
(2 rows)

test=# ---- S T E P 3 ----
test=# UPDATE accounts SET balance = balance - 100 WHERE account_id = 1;
UPDATE 1
test=# SELECT * FROM accounts;
 account_id | balance 
-----+-----
          2 |      500
          1 |      400
(2 rows)

test=# ---- S T E P 5 ----
test=# UPDATE accounts SET balance = balance + 100 WHERE account_id = 2;
UPDATE 1
test=# COMMIT;
COMMIT
test=# SELECT * from accounts;
 account_id | balance 
-----+-----
          1 |      400
          2 |      600
(2 rows)

test=# COMMIT;
COMMIT
test=#

test=# ---- S T E P 2 ----
test=# BEGIN TRANSACTION;
BEGIN
test=# SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SET
test=# SELECT * FROM accounts;
 account_id | balance 
-----+-----
          1 |      500
          2 |      500
(2 rows)

test=# ---- S T E P 4 ----
test=# SELECT balance from accounts WHERE account_id = 2;
 balance 
-----
      500
(1 row)

test=# ---- S T E P 6 ----
test=# SELECT balance from accounts WHERE account_id = 1;
 balance 
-----
      500
(1 row)

test=# COMMIT;
COMMIT
test=#

```

Figure 4.16: Read skew in Postgres's SERIALIZABLE isolation level

PostgreSQL's SERIALIZABLE isolation level provides the same consistent read as the REPEATABLE READ isolation level does. This means that it will behave the same when encountering a read skew anomaly, which is to never allow it to happen.

4.3.16 Write skew and REPEATABLE READ

To examine the write skew anomaly, the SQL code below is used.

The code creates a hospital table with doctors and their "on call" status.

Note: The hospital database model has a constraint requiring that **at least one** doctor must always be on call.

```
CREATE TABLE doctors (name VARCHAR(10),
                      on_call VARCHAR(10), shift_id INT);
INSERT INTO doctors (name, on_call, shift_id)
VALUES ('Alice', 'true', 1234);
INSERT INTO doctors (name, on_call, shift_id)
VALUES ('Bob', 'true', 1234);
INSERT INTO doctors (name, on_call, shift_id)
VALUES ('Carol', 'false', 1234);
CREATE FUNCTION currently_on_call() RETURNS BIGINT AS \${$}
SELECT COUNT(*)
FROM doctors where on_call = 'true' and shift_id=1234; \${$}
LANGUAGE SQL
```

Figure 4.17 below illustrates a case where Alice and Bob apply for a leave simultaneously.

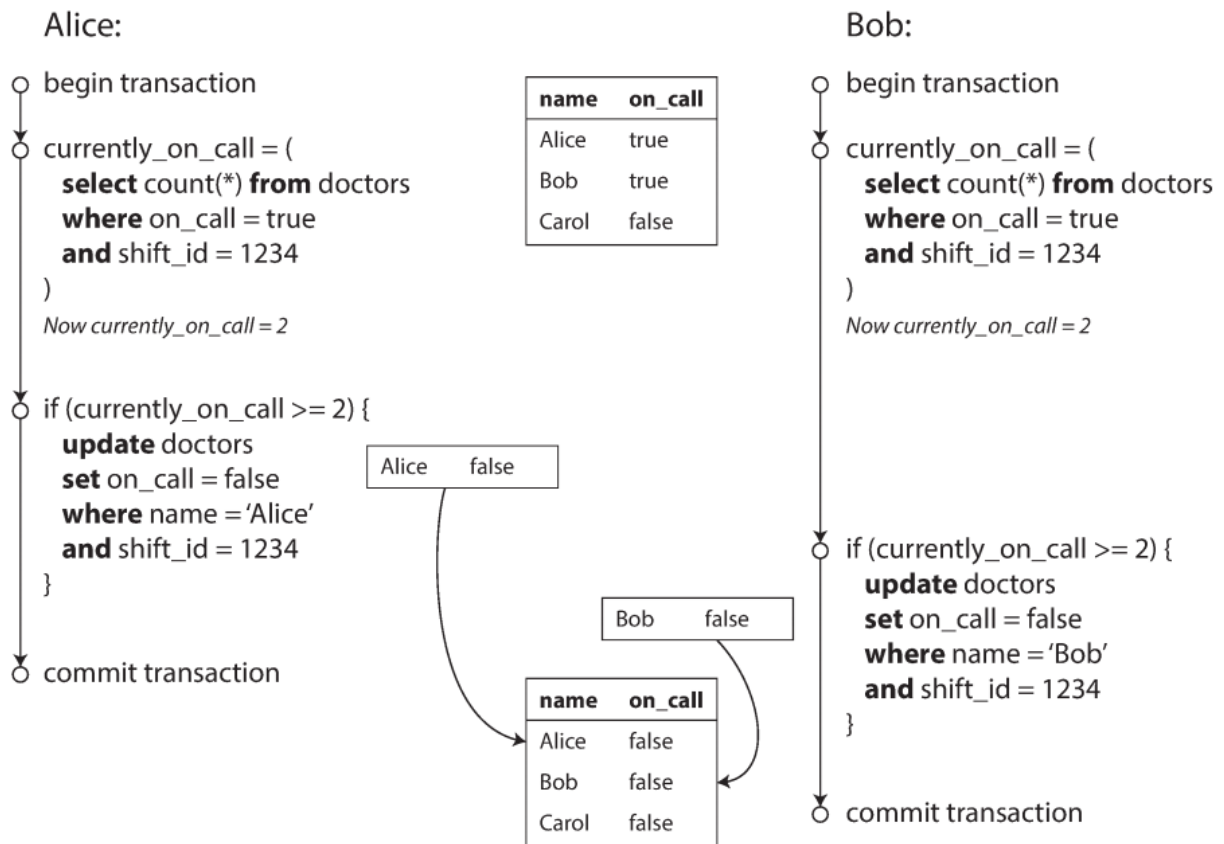


Figure 4.17: Example of Write Skew occurring (all false) [2]

SESSION A

SESSION B

<pre> test=# ---- S T E P 1 ---- test=# BEGIN TRANSACTION; test=# SET TRANSACTION ISOLATION LEVEL REPEATABLE READ; test=# SET test=# SELECT * FROM doctors ORDER BY name ASC; name on_call shift_id -----+-----+----- Alice true 1234 Bob true 1234 Carol false 1234 (3 rows) test=# ---- S T E P 3 ---- test=# UPDATE doctors SET on_call = CASE WHEN ((SELECT currently_on_cal\$ ()) >= 2) THEN 'false' ELSE on_call END WHERE name = 'Alice' and shift_id\$ 1234; UPDATE 1 test=# COMMIT; test=# COMMIT; test=# SELECT * FROM doctors ORDER BY name ASC; name on_call shift_id -----+-----+----- Alice false 1234 Bob true 1234 Carol false 1234 (3 rows) test=# COMMIT; test=# COMMIT; </pre>	<pre> test=# ---- S T E P 2 ---- test=# BEGIN TRANSACTION; test=# SET TRANSACTION ISOLATION LEVEL REPEATABLE READ; test=# SET test=# SELECT * FROM doctors ORDER BY name ASC; name on_call shift_id -----+-----+----- Alice true 1234 Bob true 1234 Carol false 1234 (3 rows) test=# ---- S T E P 4 ---- test=# UPDATE doctors SET on_call = CASE WHEN ((SELECT currently_on_cal\$ ()) >= 2) THEN 'false' ELSE on_call END WHERE name = 'Bob' and shift_id\$ 1234; UPDATE 1 test=# COMMIT; test=# COMMIT; test=# SELECT * FROM doctors ORDER BY name ASC; name on_call shift_id -----+-----+----- Alice false 1234 Bob false 1234 Carol false 1234 (3 rows) test=# COMMIT; test=# COMMIT; </pre>
--	--

Figure 4.18: Write skew in PostgreSQL's REPEATABLE READ isolation level

Write skew anomalies are present in the REPEATABLE READ isolation level. PostgreSQL's REPEATABLE READ isolation level is no exception to the rule, as it is prone to write skew phenomena.

In Step 4 of Session B we note that the change at Step 3 of Session A, which changed Alice's `on_call` status to false, is not seen and as a result the count of `doctors_on_call` remains 2. So, Step 4 succeeds in updating Bob's `on_call` status to 'false'. After Session B commits at Step 4, we see that now all doctors are out of call. This violates the hospital's model constraint mentioned above.

4.3.17 Write skew and SERIALIZABLE

SESSION A

SESSION B

```

POSTGRES - TEST18 - # Anomaly|Write Skew - Isolation|Serializable

File Edit View Terminal Tabs Help

test=# ---- S T E P 1 ----
test=# BEGIN TRANSACTION;
test=# SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
test=# SET
test=# SELECT * FROM doctors ORDER BY name ASC;
name | on_call | shift_id
-----+-----+-----
Alice | true    | 1234
Bob   | true    | 1234
Carol | false   | 1234
(3 rows)

test=# ---- S T E P 3 ----
test=# UPDATE doctors SET on_call = CASE WHEN ( (SELECT currently_on_call$
()) >= 2) THEN 'false' ELSE on_call END WHERE name = 'Alice' and shift_id$
1234;
UPDATE 1
test=# COMMIT;
SELECT * FROM doctors ORDER BY name ASC;
name | on_call | shift_id
-----+-----+-----
Alice | false   | 1234
Bob   | true    | 1234
Carol | false   | 1234
(3 rows)

test=# COMMIT;
test=#

test=# ---- S T E P 2 ----
test=# BEGIN TRANSACTION;
test=# SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
test=# SET
test=# SELECT * FROM doctors ORDER BY name ASC;
name | on_call | shift_id
-----+-----+-----
Alice | true    | 1234
Bob   | true    | 1234
Carol | false   | 1234
(3 rows)

test=# ---- S T E P 4 ----
test=# UPDATE doctors SET on_call = CASE WHEN ( (SELECT currently_on_call$
)) >= 2) THEN 'false' ELSE on_call END WHERE name = 'Bob' and shift_id=123$
;
ERROR:  could not serialize access due to read/write dependencies among tr$
nsactions
DETAIL:  Reason code: Canceled on identification as a pivot, during write.
HINT:  The transaction might succeed if retried.
test=# COMMIT;
ROLLBACK
test=# SELECT * FROM doctors ORDER BY name ASC;
name | on_call | shift_id
-----+-----+-----
Alice | false   | 1234
Bob   | true    | 1234
Carol | false   | 1234
(3 rows)

test=# COMMIT;
test=#

```

Figure 4.19: Write skew in PostgreSQL's SERIALIZABLE isolation level

PostgreSQL's SERIALIZABLE isolation level, prevents write skew anomalies while at the same time maintaining REPEATABLE READ's benefits.

As one can see, in Step 4 of Session B, even though the doctors on call seem to be 2, the update is not allowed and a *"could not serialize access due to read/write dependencies among transactions"* exception is thrown. This ensures that the hospital's constraint of **at least one** doctor being on call is not violated.

4.3.18 Write skew and READ COMMITTED

SESSION A

SESSION B

```

POSTGRES - TEST16 - # Anomaly|Write Skew - Isolation|Read Committed

File Edit View Terminal Tabs Help

test=# ---- S T E P 1 ----
test=# BEGIN TRANSACTION;
test=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
test=# SET
test=# SELECT * FROM doctors ORDER BY name ASC;
name | on_call | shift_id
-----+-----+-----
Alice | true    | 1234
Bob   | true    | 1234
Carol | false   | 1234
(3 rows)

test=# ---- S T E P 3 ----
test=# UPDATE doctors SET on_call = CASE WHEN ( (SELECT currently_on_cal$
()) >= 2) THEN 'false' ELSE on_call END WHERE name = 'Alice' and shift_id$
1234;
COMMIT;UPDATE 1
test=# COMMIT;
COMMIT
test=# SELECT * FROM doctors ORDER BY name ASC;
name | on_call | shift_id
-----+-----+-----
Alice | false   | 1234
Bob   | true    | 1234
Carol | false   | 1234
(3 rows)

test=# COMMIT;
COMMIT
test=#

test=# ---- S T E P 2 ----
test=# BEGIN TRANSACTION;
test=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
test=# SET
test=# SELECT * FROM doctors ORDER BY name ASC;
name | on_call | shift_id
-----+-----+-----
Alice | true    | 1234
Bob   | true    | 1234
Carol | false   | 1234
(3 rows)

test=# ---- S T E P 4 ----
test=# UPDATE doctors SET on_call = CASE WHEN ( (SELECT currently_on_cal$
)) >= 2) THEN 'false' ELSE on_call END WHERE name = 'Bob' and shift_id=123$
;
UPDATE 1
test=# COMMIT;
SELECT * FROM doctors ORDER BY name ASC;
COMMIT
test=# SELECT * FROM doctors ORDER BY name ASC;
name | on_call | shift_id
-----+-----+-----
Alice | false   | 1234
Bob   | true    | 1234
Carol | false   | 1234
(3 rows)

test=# COMMIT;
COMMIT
test=#

```

Figure 4.20: Write skew in Postgres's READ COMMITTED isolation level

The READ COMMITTED isolation level is not prone to write skew phenomena. We can observe the expected behaviour in PostgreSQL's READ COMMITTED isolation level example. Session B on Step 4 will return "1" when `currently_on_call` is called, so the UPDATE query will not set Bob's `on_call` status to "false".

4.4 Comparison of test results

This table sums up the findings of how PostgreSQL isolation levels behave when encountering the tested transaction anomalies.

Table 4.1: PostgreSQL isolation levels and transaction anomalies

	Read Committed	Repeatable Read	Serializable
NON-REPEATABLE READ	Possible	Not Possible	Not Possible
LOST UPDATE	Possible	Not Possible ¹	Not Possible ¹
GHOST UPDATE	Not Possible ²	Not Possible ¹	Not Possible ¹
PHANTOM UPDATE	Possible	Not Possible ²	Not Possible ²
READ SKEW	Possible	Not Possible	Not Possible
WRITE SKEW	Not Possible	Possible	Not Possible ²

¹ Exception occurs and transaction rolls back

² No exception, transaction continues

Chapter 5

MVCC in SQL Server

5.1 SQL Server Transaction Isolation Levels

SQL Server implements all four isolation levels that are defined in the ANSI SQL '92 standard. Two additional isolation levels are implemented using **row versioning**. The one is the statement-level snapshot isolation equivalent of READ COMMITTED, which SQL Server calls Read Committed Snapshot Isolation (**RCSI**). The other one provides transaction-level read consistency and it is called SNAPSHOT, often referred to as Snapshot Isolation (**SI**). This thesis is going to focus on those two isolation levels since only they use MVCC technology under the hood.

5.1.1 READ COMMITTED with Snapshot Isolation

This isolation level is oftentimes called **RCSI**. The reason is that, in order to use it, one must first set the `READ_COMMITTED_SNAPSHOT` database setting to `ON` [22]. By default, this option is set to `OFF`, which means that if someone sets the transaction isolation level to `READ COMMITTED`, they will be using the pessimistic locking implementation of `READ COMMITTED`. Both regular `READ COMMITTED` and **RCSI**, see only committed data. The main difference lies in the fact that **RCSI** doesn't cause blocking.

SQL Server stores the data of a row, in `tempdb`, before they get modified. In that way, any operation that relies on a previous version of data, will be able to access them. *Exclusive locks are still acquired by update transactions but, because we now have a previous committed version of the row, a concurrent read transaction can proceed without being blocked. Each query will read the database at a consistent point-in-time using row versions. If a query encounters a row that has been updated since the query began, the previous version of the row is read from the `tempdb` [23].* Even though there is the convenience of not needing to handle exceptions in this isolation level, developers must be wary of the potential anomalies that might occur. These include non-repeatable reads, lost updates, phantom updates and skewed reads.

5.1.2 SNAPSHOT

Snapshot Isolation (**SI**) is the second isolation level that implements MVCC in SQL Server. *The term "snapshot" reflects the fact that all queries in the transaction see the same version, or snapshot, of the database, based on the state of the database at the moment in time when the transaction begins [24].* Changes made by other transactions after the transaction began will not be visible but changes made by the transaction itself are visible to all its subsequent queries.

Database setting `ALLOW_SNAPSHOT_ISOLATION` must be set to `ON` before one is able to use this isolation level. One important characteristic of **SNAPSHOT** is that if a transaction attempts to commit modifications to data that have changed since the transaction began, the transaction will roll back and an error will be raised [24]. Such errors will not be encountered when running **RCSI**. In terms of anomaly prevention, this isolation level is susceptible only to skewed writes.

5.2 SQL Server and MVCC

Row versions (logical copies) are kept for all data before they get modified in the database, when either `READ_COMMITTED_SNAPSHOT` or `ALLOW_SNAPSHOT_ISOLATION` is set to `ON`. Copies are kept, even if there are not any active transactions using a row versioning-based isolation level.

These copies are kept in SQL Server's **tempdb**. Every time a row is **modified** by a transaction, SQL Server stores the image of the previously committed version in tempdb and stamps it with the transaction's sequence number (XSN). *The versions of modified rows are chained using a link list. The newest row value is always stored in the current database and chained to the versioned rows stored in tempdb* [22].

Row versions must be stored for as long as an active transaction needs to access them. In order to free up space in tempdb, a background operation removes rows that are no longer needed by active transactions. This operation tracks the earliest useful XSN and periodically deletes all row versions stamped with XSNs that are lower than the earliest useful one.

In `SNAPSHOT` isolation level, every time a row is **read**, SQL Server follows the chain and retrieves the row where the XSN is:

- *Closest to, but lower than, the XSN of the snapshot transaction reading the row*
- *Not in the list of the transactions that were active when the snapshot transaction started* [22]

The `READ COMMITTED` with snapshot isolation (RCSI) works almost the same way. The difference is that a transaction using RCSI will not use its own XSN for retrieving versions of rows from tempdb. It will use the latest (committed) XSN that exists before the current statement gets executed. This is to guarantee statement level consistency.

5.3 Examples of isolation levels

The following scenarios are being tested in two, side-by-side active concurrent transactions. The table that is being used got created with this SQL code:

```
CREATE TABLE T (id INT NOT NULL PRIMARY KEY, x INT);
INSERT INTO T VALUES (1, 1);
INSERT INTO T VALUES (2, 2);
INSERT INTO T VALUES (3, 3);
```

Every scenario using the READ COMMITTED isolation level has previously enabled the READ_COMMITTED_SNAPSHOT setting, as well as the ALLOW_SNAPSHOT_ISOLATION in order to be used with the SNAPSHOT isolation level.

5.3.1 Non-repeatable read and READ COMMITTED

SESSION A

SESSION B

```

SQLSERVER - TEST1 - # Anomaly|Non-Repeatable Read - Isolation|Read Committed

File Edit View Terminal Tabs Help

2> -- S T E P 1 --
3> GO
1> -----
2> GO
1> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
2> GO
1> BEGIN TRANSACTION;
2> GO
1> SELECT * FROM T WHERE id=1;
2> GO
id      x
-----
1      1

(1 rows affected)
1> -- S T E P 3 --
2> GO
1> -----
2> GO
1> SELECT * FROM T ORDER BY id;
2> GO
id      x
-----
1      1
2      2
3      3

(3 rows affected)
1> -- S T E P 5 --
2> GO
1> -----
2> GO
1> SELECT * FROM T ORDER BY id;
2> GO
id      x
-----
1     12
2      2
3      3

(3 rows affected)
1> COMMIT;
2> GO
1>

Btech123 -d T -S 127.0.0.1
1> SET IMPLICIT_TRANSACTIONS ON
2> :RESET
1>
2> -- S T E P 2 --
3> GO
1> -----
2> GO
1> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
2> GO
1> BEGIN TRANSACTION;
2> GO
1> UPDATE T SET x=12 WHERE id=1;
2> GO

(1 rows affected)
1> SELECT * FROM T WHERE id=1;
2> GO
id      x
-----
1     12

(1 rows affected)
1> -- S T E P 4 --
2> GO
1> -----
2> GO
1> SELECT * FROM T WHERE id=1;
2> GO
id      x
-----
1     12

(1 rows affected)
1> COMMIT;
2> GO
1>

```

Figure 5.1: Non-repeatable read in SQL Server's READ COMMITTED isolation level

The example above demonstrates that SQL Server's READ COMMITTED isolation level does not provide a repeatable read, as Session A on Step 5 shows the changed value that the

Session B committed in Step 4. Also, a dirty read is not allowed as Session A on Step 3 would be able to see the changed value if that was the case.

5.3.2 Non-repeatable read and SNAPSHOT

SESSION A

SESSION B

```

SQLSERVER - TEST2 - # Anomaly|Non-Repeatable Read - Isolation|Snapshot

File Edit View Terminal Tabs Help

2> -- S T E P 1 --
3> GO
1> -----
2> GO
1> SET TRANSACTION ISOLATION LEVEL SNAPSHOT;
2> GO
1> BEGIN TRANSACTION;
2> GO
1> SELECT * FROM T WHERE id=1;
2> GO
id          x
-----
1          1

(1 rows affected)
1> -- S T E P 3 --
2> GO
1> -----
2> GO
1> SELECT * FROM T WHERE id=1;
2> GO
id          x
-----
1          1

(1 rows affected)
1> COMMIT;
2> GO
1> SELECT * FROM T ORDER BY id;
2> GO
id          x
-----
1          12
2           2
3           3

(3 rows affected)
1>

Btech123 -d T -S 127.0.0.1
1> SET IMPLICIT_TRANSACTIONS ON
2> :RESET
1>
2> -- S T E P 2 --
3> GO
1> -----
2> GO
1> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
2> GO
1> BEGIN TRANSACTION;
2> GO
1> UPDATE T SET x=12 WHERE id=1;
2> GO

(1 rows affected)
1> SELECT * FROM T WHERE id=1;
2> GO
id          x
-----
1          12

(1 rows affected)
1> COMMIT;
2> GO
1>

```

Figure 5.2: Non-repeatable read in SQL Server's SNAPSHOT isolation level

SNAPSHOT isolation level provides repeatable reads.

One can observe that Session A on Step 3 sees the table value unchanged, even after Session B has committed on Step 2.

5.3.3 Lost update and READ COMMITTED

SESSION A

SESSION B

```

SQLSERVER - TEST3 - # Anomaly|Lost Update - Isolation|Read Committed

File Edit View Terminal Tabs Help

2> -- S T E P 1 --
3> GO
1> -----
2> GO
1> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
2> GO
1> BEGIN TRANSACTION;
2> GO
1> SELECT * FROM T WHERE id=1;
2> GO
id          x
-----
1          1

(1 rows affected)
1> -- S T E P 3 --
2> GO
1> -----
2> GO
1> UPDATE T SET x=13 WHERE id=1;
2> GO

(1 rows affected)
1> SELECT * FROM T WHERE id=1;
2> GO
id          x
-----
1          13

(1 rows affected)
1> COMMIT;
2> GO
1> SELECT * FROM T ORDER BY id;
2> GO
id          x
-----
1          13
2           2
3           3

(3 rows affected)
1>

Btech123 -d T -S 127.0.0.1
1> SET IMPLICIT_TRANSACTIONS ON
2> :RESET
1>
2> -- S T E P 2 --
3> GO
1> -----
2> GO
1> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
2> GO
1> BEGIN TRANSACTION;
2> GO
1> UPDATE T SET x=12 WHERE id=1;
2> GO

(1 rows affected)
1> SELECT * FROM T WHERE id=1;
2> GO
id          x
-----
1          12

(1 rows affected)
1> COMMIT;
2> GO
1>

```

Figure 5.3: Lost update in SQL Server's READ COMMITTED isolation level

Here one sees the lost update phenomenon occurring in SQL Server's READ COMMITTED isolation level. Session B's update ($x=12$, on Step 2) gets overwritten by Session A's update ($x=13$, on Step 3), without a warning or an exception.

5.3.4 Lost update and SNAPSHOT

SESSION A

SESSION B

The screenshot shows a SQL Server window titled "SQLSERVER - TEST4 - # Anomaly|Lost Update - Isolation|Snapshot". It displays two sessions, A and B, with their respective SQL commands and results.

Session A:

```

2> -- S T E P 1 --
3> GO
1> -----
2> GO
1> SET TRANSACTION ISOLATION LEVEL SNAPSHOT;
2> GO
1> BEGIN TRANSACTION;
2> GO
1> SELECT * FROM T WHERE id=1;
2> GO
id          x
-----
1          1
(1 rows affected)
1> -- S T E P 3 --
2> GO
1> -----
2> GO
1> UPDATE T SET x=13 WHERE id=1;
2> GO
Msg 3960, Level 16, State 3, Server debian10, Line 1
Snapshot isolation transaction aborted due to update conflict. You cannot
use snapshot isolation to access table 'dbo.T' directly or indirectly in d
atabase 'T' to update, delete, or insert the row that has been modified or
deleted by another transaction. Retry the transaction or change the isola
tion level for the update/delete statement.
1> SELECT * FROM T ORDER BY id;
2> GO
id          x
-----
1          12
2           2
3           3
(3 rows affected)
1>

```

Session B:

```

Btech123 -d T -S 127.0.0.1
1> SET IMPLICIT_TRANSACTIONS ON
2> :RESET
1>
2> -- S T E P 2 --
3> GO
1> -----
2> GO
1> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
2> GO
1> BEGIN TRANSACTION;
2> GO
1> UPDATE T SET x=12 WHERE id=1;
2> GO
(1 rows affected)
1> SELECT * FROM T WHERE id=1;
2> GO
id          x
-----
1          12
(1 rows affected)
1> COMMIT;
2> GO
1>

```

Figure 5.4: Lost update in SQL Server's SNAPSHOT isolation level

In a SNAPSHOT isolation environment, an exception is thrown the moment a lost update phenomenon is about to occur. This can be seen in Session A, where Step 3 tries to update a tuple that got updated and committed by Session B on Step 2. An exception *"Snapshot isolation transaction aborted due to update conflict"* is thrown. Session A's update ($x = 13$, on Step 3) is not completed and the transaction is automatically rolled back. The next `SELECT` statement demonstrates that Session A (after the rollback) can see Session B's committed change ($x = 12$).

5.3.5 Ghost update and READ COMMITTED

SESSION A

SESSION B

```

SQLSERVER - TEST5 - # Anomaly|Ghost Update - Isolation|Read Committed

File Edit View Terminal Tabs Help

2> -- S T E P 1 --
3> GO
1> -----
2> GO
1> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
2> GO
1> BEGIN TRANSACTION;
2> GO
1> SELECT * FROM T ORDER BY id;
2> GO
id      x
-----
1      1
2      2
3      3
(3 rows affected)
1> -- S T E P 3 --
2> GO
1> -----
2> GO
1> UPDATE T SET x=13 WHERE id=1;
2> GO
(0 rows affected)
1> COMMIT;
2> GO
1> SELECT * FROM T ORDER BY id;
2> GO
id      x
-----
2      2
3      3
(2 rows affected)
1>

Btech123 -d T -S 127.0.0.1
1> SET IMPLICIT_TRANSACTIONS ON
2> :RESET
1>
2> -- S T E P 2 --
3> GO
1> -----
2> GO
1> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
2> GO
1> BEGIN TRANSACTION;
2> GO
1> DELETE FROM T WHERE id=1;
2> GO
(1 rows affected)
1> SELECT * FROM T ORDER BY id;
2> GO
id      x
-----
2      2
3      3
(2 rows affected)
1> COMMIT;
2> GO
1>

```

Figure 5.5: Ghost update in SQL Server's READ COMMITTED isolation level

The ghost update that was about to happen, in Session A on Step 3, did not occur and the `UPDATE` query resulted in *(0 rows affected)*. This is because it was previously deleted in Session B on Step 2. Notably, no exception or rollback took place, and the transaction continued. Ghost update phenomena do not take place in SQL Server's READ COMMITTED isolation level.

5.3.6 Ghost update and SNAPSHOT

SESSION A

SESSION B

```

SQLSERVER - TEST6 - # Anomaly|Ghost Update - Isolation|Snapshot

File Edit View Terminal Tabs Help

2> -- S T E P 1 --
3> GO
1> -----
2> GO
1> SET TRANSACTION ISOLATION LEVEL SNAPSHOT;
2> GO
1> BEGIN TRANSACTION;
2> GO
1> SELECT * FROM T ORDER BY id;
2> GO
id      x
-----
1      1
2      2
3      3

(3 rows affected)
1> -- S T E P 3 --
2> GO
1> -----
2> GO
1> UPDATE T SET x=13 WHERE id=1;
2> GO
Msg 3960, Level 16, State 3, Server debian10, Line 1
Snapshot isolation transaction aborted due to update conflict. You cannot
use snapshot isolation to access table 'dbo.T' directly or indirectly in d
atabase 'T' to update, delete, or insert the row that has been modified or
deleted by another transaction. Retry the transaction or change the isola
tion level for the update/delete statement.
1> COMMIT;
2> GO
Msg 3902, Level 16, State 1, Server debian10, Line 1
The COMMIT TRANSACTION request has no corresponding BEGIN TRANSACTION.
1> SELECT * FROM T ORDER BY id;
2> GO
id      x
-----
2      2
3      3

(2 rows affected)
1>

Btech123 -d T -S 127.0.0.1
1> SET IMPLICIT_TRANSACTIONS ON
2> :RESET
1>
2> -- S T E P 2 --
3> GO
1> -----
2> GO
1> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
2> GO
1> BEGIN TRANSACTION;
2> GO
1> DELETE FROM T WHERE id=1;
2> GO

(1 rows affected)
1> SELECT * FROM T ORDER BY id;
2> GO
id      x
-----
2      2
3      3

(2 rows affected)
1> COMMIT;
2> GO
1>

```

Figure 5.6: Ghost update in SQL Server's SNAPSHOT isolation level

The REPEATABLE READ isolation level is prone to ghost update anomalies, but SQL Server's SNAPSHOT prevents them from happening by stopping the execution with a message **transaction aborted due to update conflict** (Session A Step 3) and rolling back the transaction automatically.

5.3.7 Phantom update and READ COMMITTED

SESSION A

SESSION B

```

SQLSERVER - TEST7 - # Anomaly|Phantom Update - Isolation|Read Committed
File Edit View Terminal Tabs Help

2> -- S T E P 1 --
3> GO
1> -----
2> GO
1> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
2> GO
1> BEGIN TRANSACTION;
2> GO
1> SELECT * FROM T ORDER BY id;
2> GO
id      x
-----
      1      1
      2      2
      3      3

(3 rows affected)
1> -- S T E P 3 --
2> GO
1> -----
2> GO
1> UPDATE T SET x=42 WHERE id=4;
2> GO

(1 rows affected)
1> SELECT * FROM T ORDER BY id;
2> GO
id      x
-----
      1      1
      2      2
      3      3
      4     42

(4 rows affected)
1> COMMIT;
2> GO
1>

md -U SA -PDBtech123 -d T -S 127.0.0.1
1> SET IMPLICIT_TRANSACTIONS ON
2> :RESET
1>
2> -- S T E P 2 --
3> GO
1> -----
2> GO
1> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
2> GO
1> BEGIN TRANSACTION;
2> GO
1> INSERT INTO T VALUES (4,4);
2> GO

(1 rows affected)
1> SELECT * FROM T ORDER BY id;
2> GO
id      x
-----
      1      1
      2      2
      3      3
      4      4

(4 rows affected)
1> COMMIT;
2> GO
1>

```

Figure 5.7: Phantom update in SQL Server's READ COMMITTED isolation level

Phantom updates are not prevented in SQL Server's READ COMMITTED isolation level. This can be seen in Session A on Step 3 where it successfully updates the new row that got inserted (and committed) by Session B on Step 2.

5.3.8 Phantom update and SNAPSHOT

SESSION A

SESSION B

```

SQLSERVER - TEST8 - # Anomaly|Phantom Update - Isolation|Snapshot

File Edit View Terminal Tabs Help

2> -- S T E P 1 --
3> GO
1> -----
2> GO
1> SET TRANSACTION ISOLATION LEVEL SNAPSHOT;
2> GO
1> BEGIN TRANSACTION;
2> GO
1> SELECT * FROM T ORDER BY id;
2> GO
id      x
-----
1      1
2      2
3      3

(3 rows affected)
1> -- S T E P 3 --
2> GO
1> -----
2> GO
1> UPDATE T SET x=42 WHERE id=4;
2> GO
Msg 3960, Level 16, State 3, Server debian10, Line 1
Snapshot isolation transaction aborted due to update conflict. You cannot
use snapshot isolation to access table 'dbo.T' directly or indirectly in d
atabase 'T' to update, delete, or insert the row that has been modified or
deleted by another transaction. Retry the transaction or change the isola
tion level for the update/delete statement.
1> COMMIT;
2> GO
Msg 3902, Level 16, State 1, Server debian10, Line 1
The COMMIT TRANSACTION request has no corresponding BEGIN TRANSACTION.
1>

Btech123 -d T -S 127.0.0.1
1> SET IMPLICIT_TRANSACTIONS ON
2> :RESET
1>
2> -- S T E P 2 --
3> GO
1> -----
2> GO
1> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
2> GO
1> BEGIN TRANSACTION;
2> GO
1> INSERT INTO T VALUES (4,4);
2> GO

(1 rows affected)
1> SELECT * FROM T ORDER BY id;
2> GO
id      x
-----
1      1
2      2
3      3
4      4

(4 rows affected)
1> COMMIT;
2> GO
1>

```

Figure 5.8: Phantom update in SQL Server's SNAPSHOT isolation level

In the case of the SNAPSHOT isolation level, because of its repeatable read functionality, phantom phenomena are prevented.

As seen in Session A on Step 3, in the attempt to update a tuple that did not exist in the start of a transaction, but got inserted and committed in another transaction (Session B Step 2) while the first was still active, he will be presented with *"Snapshot isolation transaction aborted due to update conflict"* error.

Session A's update ($x = 42$) is not completed and the transaction is automatically rolled back.

5.3.9 Read skew and READ COMMITTED

The read skew anomaly is going to get examined using the following table, which is a simulation of two bank accounts belonging to a user. The user is attempting to transfer money from one account to the other.

```
CREATE TABLE accounts (account_id INT NOT NULL PRIMARY KEY, balance INT);
INSERT INTO accounts VALUES (1, 500);
INSERT INTO accounts VALUES (2, 500);
```

SESSION A

SESSION B

```

SQLSERVER - TEST11 - # Anomaly|Read Skew - Isolation|Read Committed

File Edit View Terminal Tabs Help

1> ---- S T E P 1 ----
2> GO
1> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
2> GO
1> BEGIN TRANSACTION;
2> GO
1> SELECT * FROM accounts;
2> GO
account_id balance
-----
1          500
2          500

(2 rows affected)
1> ---- S T E P 3 ----
2> GO
1> UPDATE accounts SET balance = balance - 100 WHERE account_id = 1;
2> GO

(1 rows affected)
1> SELECT * FROM accounts;
2> GO
account_id balance
-----
1          400
2          500

(2 rows affected)
1> ---- S T E P 5 ----
2> GO
1> UPDATE accounts SET balance = balance + 100 WHERE account_id = 2;
2> GO

(1 rows affected)
1> COMMIT;
2> GO
1> SELECT * from accounts;
2> GO
account_id balance
-----
1          400
2          600

(2 rows affected)
1>

Btech123 -d T -S 127.0.0.1
1> SET IMPLICIT_TRANSACTIONS ON
2> :RESET
1>
2> ---- S T E P 2 ----
3> GO
1> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
2> GO
1> BEGIN TRANSACTION;
2> GO
1> SELECT * FROM accounts;
2> GO
account_id balance
-----
1          500
2          500

(2 rows affected)
1> ---- S T E P 4 ----
2> GO
1> SELECT balance from accounts WHERE account_id = 2;
2> GO
balance
-----
500

(1 rows affected)
1> ---- S T E P 6 ----
2> GO
1> SELECT balance from accounts WHERE account_id = 1;
2> GO
balance
-----
400

(1 rows affected)
1>

```

Figure 5.9: Read skew in SQL Server's READ COMMITTED isolation level

In Steps 3 and 5 of the left session (Session A) of Figure 5.9, an attempt is made to transfer **€100** from account 1 to account 2. In Step 4 of the session in the right (Session B) of Figure 5.9, the current balance of account 2 is read before Step 5 (of Session A) adds **€100** to it. The value read equals **€500**. Next, Step 6 of Session B reads the current balance of account 1, following the subtraction of **€100** in Step 3 of Session A. The value read is **€400**. As a result, the transaction in Session B has read a combined value of accounts 1 and 2 that is equal to $500 + 400 = \text{€}900$ instead of the correct value which is **€1000**. This comprises the so called read skew problem. In essence, it is a violation of the data integrity constraint

requiring the combined value of the two bank accounts to always remain equal to €1000. SQL Server's READ COMMITTED isolation level is prone to Read Skew anomalies.

5.3.10 Read skew and SNAPSHOT

SESSION A

SESSION B

```

SQLSERVER - TEST12 - # Anomaly|Read Skew - Isolation|Snapshot

File Edit View Terminal Tabs Help

1> ---- S T E P 1 ----
2> GO
1> SET TRANSACTION ISOLATION LEVEL SNAPSHOT;
2> GO
1> BEGIN TRANSACTION;
2> GO
1> SELECT * FROM accounts;
2> GO
account_id balance
-----
1          500
2          500

(2 rows affected)
1> ---- S T E P 3 ----
2> GO
1> UPDATE accounts SET balance = balance - 100 WHERE account_id = 1;
2> GO

(1 rows affected)
1> SELECT * FROM accounts;
2> GO
account_id balance
-----
1          400
2          500

(2 rows affected)
1> ---- S T E P 5 ----
2> GO
1> UPDATE accounts SET balance = balance + 100 WHERE account_id = 2;
2> GO

(1 rows affected)
1> COMMIT;
2> GO
1> SELECT * from accounts;
2> GO
account_id balance
-----
1          400
2          600

(2 rows affected)
1>

Btech123 -d T -S 127.0.0.1
1> SET IMPLICIT_TRANSACTIONS ON
2> :RESET
1>
2> ---- S T E P 2 ----
3> GO
1> SET TRANSACTION ISOLATION LEVEL SNAPSHOT;
2> GO
1> BEGIN TRANSACTION;
2> GO
1> SELECT * FROM accounts;
2> GO
account_id balance
-----
1          500
2          500

(2 rows affected)
1> ---- S T E P 4 ----
2> GO
1> SELECT balance from accounts WHERE account_id = 2;
2> GO
balance
-----
500

(1 rows affected)
1> ---- S T E P 6 ----
2> GO
1> SELECT balance from accounts WHERE account_id = 1;
2> GO
balance
-----
500

(1 rows affected)
1>

```

Figure 5.10: Read skew in SQL Server's SNAPSHOT isolation level

The SQL Server's Snapshot isolation level, due to its repeatable read functionality, is not prone to read skew anomalies. So no matter *when* Session B reads each bank account, it will consistently read 500 for each one of them.

5.3.11 Write skew and SNAPSHOT

To examine the write skew anomaly, the SQL code below is used.

The code creates a hospital table with doctors and their "on call" status.

Note: The hospital database model has a constraint requiring that **at least one** doctor must always be on call.

```
CREATE TABLE doctors (name VARCHAR(10) NOT NULL PRIMARY KEY,
                      on_call VARCHAR(10), shift_id INT);
INSERT INTO doctors (name, on_call, shift_id)
VALUES ('Alice', 'true', 1234);
INSERT INTO doctors (name, on_call, shift_id)
VALUES ('Bob', 'true', 1234);
INSERT INTO doctors (name, on_call, shift_id)
VALUES ('Carol', 'false', 1234);
CREATE FUNCTION currently_on_call() RETURNS int AS BEGIN
RETURN (SELECT COUNT(*) FROM doctors
WHERE on_call = "true" and shift_id=1234); END;
```

Figure 5.11 below illustrates a case where Alice and Bob apply for a leave simultaneously.

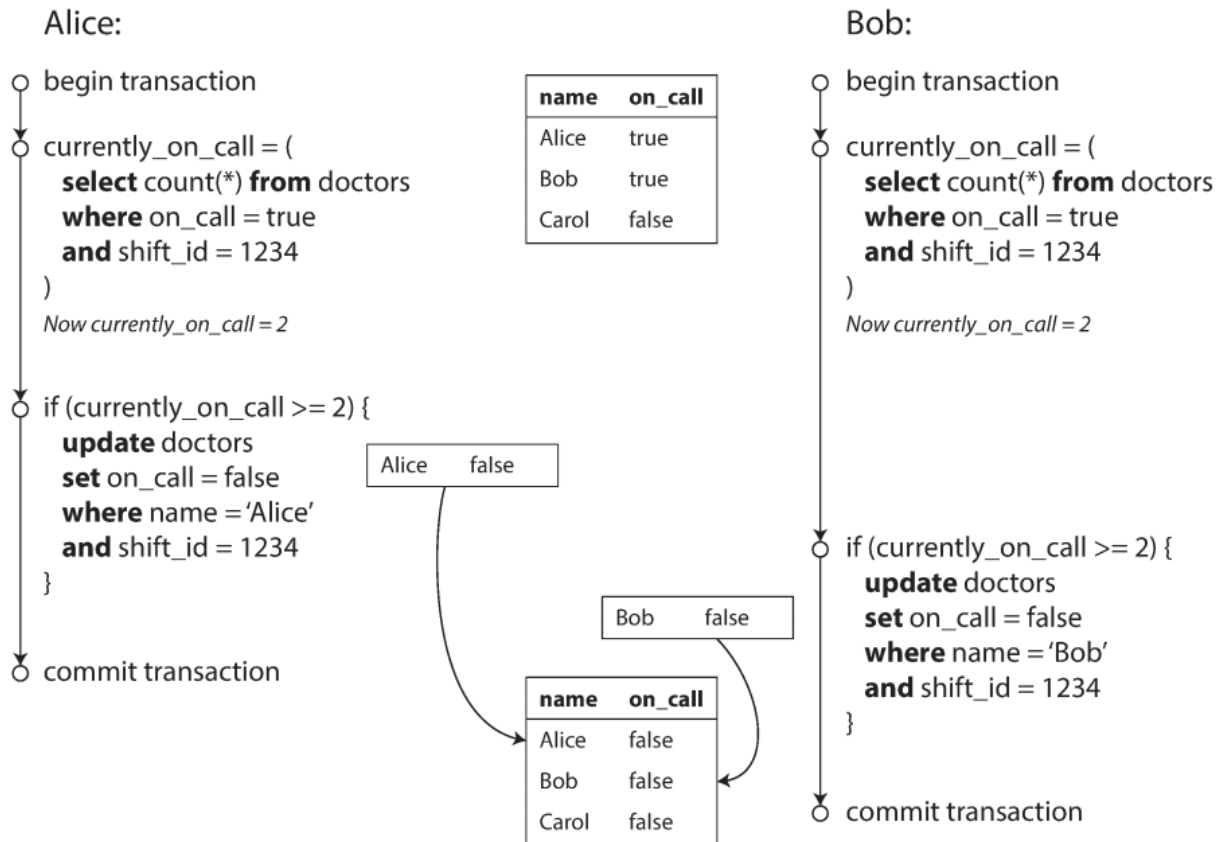


Figure 5.11: Example of write skew occurring (all false) [2]

SESSION A

SESSION B

The screenshot shows two SQL Server sessions in a window titled 'SQLSERVER - TEST10 - # Anomaly|Write Skew - Isolation|Snapshot'. The window is split into two panes, one for Session A and one for Session B.

Session A:

```

1> ---- S T E P 1 ----
2> GO
1> ALTER DATABASE T SET ALLOW_SNAPSHOT_ISOLATION ON;
2> GO
1> SET TRANSACTION ISOLATION LEVEL SNAPSHOT;
2> GO
1> BEGIN TRANSACTION;
2> GO
1> SELECT * FROM doctors;
2> GO
name      on_call  shift_id
-----
Alice     true     1234
Bob       true     1234
Carol     false    1234

(3 rows affected)
1> -- S T E P 3 --
2> GO
1> SELECT dbo.currently_on_call();
2> GO
-----
2

(1 rows affected)
1> IF dbo.currently_on_call() >= 2 UPDATE doctors SET on_call = 'false' W
HERE name = 'Alice' and shift_id=1234;
2> GO
(1 rows affected)
1> COMMIT;
2> GO
1> SELECT * FROM doctors;
2> GO
name      on_call  shift_id
-----
Alice     false    1234
Bob       true     1234
Carol     false    1234

(3 rows affected)
1>
1>
1>
1>
1>

```

Session B:

```

1> SET IMPLICIT_TRANSACTIONS ON
2> :RESET
1>
2> ---- S T E P 2 ----
3> GO
1> ALTER DATABASE T SET ALLOW_SNAPSHOT_ISOLATION ON;
2> GO
1> SET TRANSACTION ISOLATION LEVEL SNAPSHOT;
2> GO
1> SET LOCK_TIMEOUT 0;
2> GO
1> BEGIN TRANSACTION;
2> GO
1> SELECT * FROM doctors;
2> GO
name      on_call  shift_id
-----
Alice     true     1234
Bob       true     1234
Carol     false    1234

(3 rows affected)
1> -- S T E P 4 --
2> GO
1> SELECT dbo.currently_on_call();
2> GO
-----
2

(1 rows affected)
1> IF dbo.currently_on_call() >= 2 UPDATE doctors SET on_call = 'false' WH
ERE name = 'Bob' and shift_id=1234;
2> GO
(1 rows affected)
1> COMMIT;
2> GO
1> SELECT * FROM doctors;
2> GO
name      on_call  shift_id
-----
Alice     false    1234
Bob       false    1234
Carol     false    1234

(3 rows affected)
1>

```

Figure 5.12: Write skew in SQL Server's SNAPSHOT isolation level

Write skew anomalies are typically present in REPEATABLE READ isolation levels. SQL Server's SNAPSHOT isolation level, could be compared to other Database Management Systems' REPEATABLE READ isolation level.

In Step 4 of Session B we note that the change at Step 3 of Session A, which changed Alice's `on_call` status to false, is not seen and as a result the count of `doctors_on_call` remains 2. So, Step 4 succeeds in updating Bob's `on_call` status to 'false'. After Session B commits at Step 4, we see that now all doctors are out of call. This violates the hospital's model constraint mentioned above.

5.3.12 Write skew and READ COMMITTED

SESSION A

SESSION B

```

SQLSERVER - TEST9 - # Anomaly|Write Skew - Isolation|Read Committed

File Edit View Terminal Tabs Help

1> ---- S T E P 1 ----
2> GO
1> ALTER DATABASE T SET READ_COMMITTED_SNAPSHOT ON;
2> GO
1> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
2> GO
1> BEGIN TRANSACTION;
2> GO
1> SELECT * FROM doctors;
2> GO
name      on_call  shift_id
-----
Alice     true     1234
Bob       true     1234
Carol     false    1234

(3 rows affected)
1> -- S T E P 3 --
2> GO
1> SELECT dbo.currently_on_call()
2> GO
-----
2

(1 rows affected)
1> IF dbo.currently_on_call() >= 2 UPDATE doctors SET on_call = 'false' W
HERE name = 'Alice' and shift_id=1234;
2> GO

(1 rows affected)
1> GO
1> GO
1> COMMIT;
2> GO
1> SELECT * FROM doctors;
2> GO
name      on_call  shift_id
-----
Alice     false    1234
Bob       true     1234
Carol     false    1234

(3 rows affected)
1>
1>
1>

1> SET IMPLICIT_TRANSACTIONS ON
2> :RESET
1>
2> ---- S T E P 2 ----
3> GO
1> ALTER DATABASE T SET READ_COMMITTED_SNAPSHOT ON;
2> GO
1> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
2> GO
1> SET LOCK_TIMEOUT 0;
2> GO
1> BEGIN TRANSACTION;
2> GO
1> SELECT * FROM doctors;
2> GO
name      on_call  shift_id
-----
Alice     true     1234
Bob       true     1234
Carol     false    1234

(3 rows affected)
1> -- S T E P 4 --
2> GO
1> SELECT dbo.currently_on_call()
2> GO
-----
1

(1 rows affected)
1> IF dbo.currently_on_call() >= 2 UPDATE doctors SET on_call = 'false' WH
ERE name = 'Bob' and shift_id=1234;
2> GO
1> GO
1> GO
1> COMMIT;
2> GO
1> SELECT * FROM doctors;
2> GO
name      on_call  shift_id
-----
Alice     false    1234
Bob       true     1234
Carol     false    1234

(3 rows affected)
1>

```

Figure 5.13: Write skew in SQL Server's READ COMMITTED isolation level

As expected, write skew phenomena are not encountered in the READ COMMITTED isolation level. So, one can see in Session B, that Step 4 correctly sees 1 available doctor on call and as a result Bob is not allowed to take his absence leave, because the hospital must have at least 1 doctor on call.

5.4 Comparison of test results

This table sums up the findings of how SQL Server isolation levels behave when encountering the tested transaction anomalies.

Table 5.1: SQL Server isolation levels and transaction anomalies

	Read Committed	Snapshot
NON-REPEATABLE READ	Possible	Not Possible
LOST UPDATE	Possible	Not Possible ¹
GHOST UPDATE	Not Possible ²	Not Possible ¹
PHANTOM UPDATE	Possible	Not Possible ¹
READ SKEW	Possible	Not Possible
WRITE SKEW	Not Possible	Possible

¹ Exception occurs and transaction rolls back

² No exception, transaction continues

Chapter 6

Conclusion

This thesis presented Multi-Version Concurrency Control (MVCC) as a means of implementing transaction concurrency control in modern RDBMSs. A comprehensive review of the relevant literature has been conducted. Particular emphasis has been given to the literature relating to Oracle, MySQL, PostgreSQL and SQL Server, four of the most popular RDBMSs. A number of test scenarios, identical for all four RDBMSs, have been executed in order to be able to compare and contrast the way the latter implement MVCC. The results are summarized in Table 6.1.

Detailed knowledge was obtained regarding not only the general scope of the MVCC technology, but also the specifics of each one of the four RDBMSs. A software application (MVCC_sim.py) has been developed along the lines of this thesis. The application can be extended to support more RDBMS environments. It can be used as a supporting tool (e.g., to facilitate the presentation of examples like the ones considered in chapters 2-5, instead of executing one SQL command at a time).

Table 6.1: Transaction anomalies in all four RDBMSs

	NON REPEATABLE READ	LOST UPDATE	GHOST UPDATE	PHANTOM UPDATE	READ SKEW	WRITE SKEW
<u>ORACLE</u> Read Committed	Possible	Possible	Not Possible ¹	Possible	Possible	Not Possible
Serializable	Not Possible	Not Possible ²	Not Possible ²	Not Possible ¹	Not Possible	Not Possible ²
<u>MYSQL</u> Read Committed	Possible	Possible	Not Possible ¹	Possible	Possible	Not Possible
Repeatable Read	Not Possible	Possible	Not Possible ¹	Possible	Not Possible	Possible
<u>POSTGRES</u> Read Committed	Possible	Possible	Not Possible ¹	Possible	Possible	Not Possible
Repeatable Read	Not Possible	Not Possible ³	Not Possible ³	Not Possible ¹	Not Possible	Possible
Serializable	Not Possible	Not Possible ³	Not Possible ³	Not Possible ¹	Not Possible	Not Possible ³
<u>SQL SERVER</u> Read Committed ⁴	Possible	Possible	Not Possible ¹	Possible	Possible	Not Possible
Snapshot	Not Possible	Not Possible ³	Not Possible ³	Not Possible ³	Not Possible	Possible

¹ No exception, transaction continues² Exception occurs and must be handled programmatically. Otherwise, the transaction continues³ Exception occurs and transaction rolls back⁴ Using `READ_COMMITTED_SNAPSHOT ON`

References

- [1] Oracle. (2019) Transactions. Online Documentation 19. (Accessed on June 16, 2020). [Online]. Available: <https://docs.oracle.com/en/database/oracle/oracle-database/19/cncpt/data-concurrency-and-consistency.html>
- [2] M. Kleppmann, *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems*. " O'Reilly Media, Inc.", 2017.
- [3] J. Cole. (2014) The basics of the innodb undo logging and history system. (Accessed on June 16, 2020). [Online]. Available: <https://blog.jcole.us/2014/04/16/the-basics-of-the-innodb-undo-logging-and-history-system/>
- [4] I. Sarcevic. (2017) Multi version concurrency control. (Accessed on June 16, 2020). [Online]. Available: <http://morningcoffee.io/multiversion-concurrency-control.html>
- [5] J. Nelson. (2017) Practical guide to sql transaction isolation. (Accessed on June 16, 2020). [Online]. Available: <https://begriffs.com/posts/2017-08-01-practical-guide-sql-isolation.html>
- [6] Rokytsky. (2005) A not-so-very technical discussion of multi version concurrency control. (Accessed on June 16, 2020). [Online]. Available: https://firebirdsql.org/file/documentation/papers_presentations/Multi_version_concurrency_control.pdf
- [7] M. Laiho. (2020) Understanding mvcc. A DBTechNet publication in progress. (Accessed on June 16, 2020). [Online]. Available: https://drive.google.com/file/d/15FFnGDy_hZX9HoC5U2fX-dpt1NMNBUnM/view
- [8] P. A. Bernstein, V. Hadzilacos, and N. Goodman, "Concurrency control and recovery in database systems," vol. 370, 1987.
- [9] P. Cai. (2015) Database concurrency, part 1. Privy Engineering. (Accessed on June 16, 2020). [Online]. Available: <https://engineering.privy.com/2015/07/25/database-concurrency-part-1.html>
- [10] D. P. Reed, "Naming and synchronization in a decentralized computer system." Ph.D. dissertation, Massachusetts Institute of Technology, 1978.
- [11] A. Harrison. Ann harrison's reminiscences on interbase's beginnings. (Accessed on June 16, 2020). [Online]. Available: <https://firebirdsql.org/en/ann-harrison-s-reminiscences-on-interbase-s-beginnings/>

- [12] PostgreSQL. (2016) Concurrency control. (Accessed on June 16, 2020). [Online]. Available: <https://www.postgresql.org/docs/9.1/mvcc-intro.html>
- [13] Y. Wu, J. Arulraj, J. Lin, R. Xian, and A. Pavlo, “An empirical evaluation of in-memory multi-version concurrency control,” *Proceedings of the VLDB Endowment*, vol. 10, no. 7, pp. 781–792, 2017.
- [14] Oracle. (2019) Data concurrency and consistency. Online Documentation 19. (Accessed on June 16, 2020). [Online]. Available: <https://docs.oracle.com/en/database/oracle/oracle-database/19/cncpt/data-concurrency-and-consistency.html>
- [15] Oracle. (2017) Transactions. Online Documentation 12. (Accessed on June 16, 2020). [Online]. Available: <https://docs.oracle.com/database/121/CNCPT/transact.htm>
- [16] J. Bodnar. (2017, Jan.) Mysql storage engines. (Accessed on June 16, 2020). [Online]. Available: <http://zetcode.com/databases/mysqltutorial/storageengines/>
- [17] MySQL. (2020) Consistent nonlocking reads. Reference Manual 8.0. (Accessed on June 16, 2020). [Online]. Available: <https://dev.mysql.com/doc/refman/8.0/en/innodb-consistent-read.html>
- [18] MySQL. (2020) Transaction isolation levels. Reference Manual 8.0. (Accessed on June 16, 2020). [Online]. Available: <https://dev.mysql.com/doc/refman/8.0/en/innodb-transaction-isolation-levels.html>
- [19] MySQL. (2020) Inno db multi-versioning. Reference Manual 8.0. (Accessed on June 16, 2020). [Online]. Available: <https://dev.mysql.com/doc/refman/8.0/en/innodb-multi-versioning.html>
- [20] PostgreSQL. (2019) Concurrency control, transaction isolation. (Accessed on June 16, 2020). [Online]. Available: <https://www.postgresql.org/docs/12/transaction-iso.html>
- [21] E. Rogov. (2019, Sep.) Mvcc in postgresql-1. isolation. (Accessed on June 16, 2020). [Online]. Available: <https://habr.com/en/company/postgrespro/blog/467437/>
- [22] Microsoft. (2020) Transaction locking and row versioning guide. (Accessed on June 16, 2020). [Online]. Available: <https://docs.microsoft.com/en-us/sql/relational-databases/sql-server-transaction-locking-and-row-versioning-guide>
- [23] Microsoft. (2018) Sql server: Concurrency control models, acid properties and transaction isolation levels. TechNet. (Accessed on June 16, 2020). [Online]. Available: <https://social.technet.microsoft.com/wiki/contents/articles/51484.sql-server-concurrency-control-models-acid-properties-and-transaction-isolation-levels.aspx>
- [24] Microsoft. (2017) Snapshot isolation in sql server. (Accessed on June 16, 2020). [Online]. Available: <https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/sql/snapshot-isolation-in-sql-server>

Appendix A

MVCC_sim User Guide

Prerequisites

- Debian GNU/Linux 10
- Oracle Database 11g Express Edition Release 11.2.0.2.0
- MariaDB Version 15.1 Distrib 10.3.15
- PostgreSQL Version 11.5
- Microsoft SQL Server 2017 Version 14.0
- Python 2.7

Installation

The accompanying application can be downloaded from <https://github.com/kdmnt/mvcc>. The main executable file is `MVCC_sim.py` and comes along with `mvcc_runner.py` and `mvcc_tests.yml`. There is no installation needed, the first execution of `MVCC_sim.py` will install all the necessary libraries and python modules.

Usage

Navigate to the directory of the downloaded files and simply execute the following command:

```
$ python MVCC_sim.py
```

The following screen is displayed:

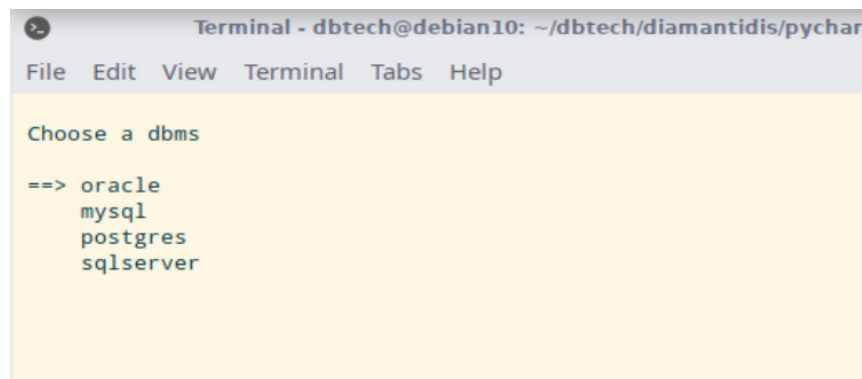
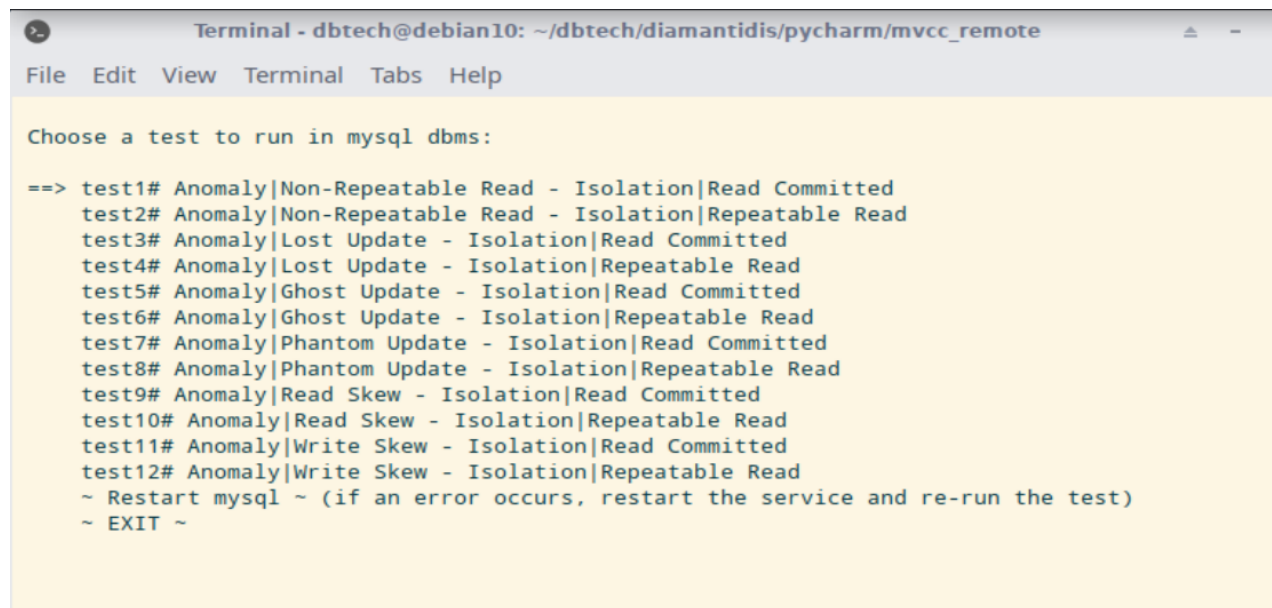


Figure A.1: Initial screen where user can select their preferred DBMS

After selecting the preferred DBMS (assuming its service is running), the next screen is displayed:

A terminal window titled "Terminal - dbtech@debian10: ~/dbtech/diamantidis/pycharm/mvcc_remote" with a menu bar containing "File", "Edit", "View", "Terminal", "Tabs", and "Help". The terminal content shows a prompt "Choose a test to run in mysql dbms:" followed by a list of 12 tests. Each test is represented by a number and a string of pipe-separated values: "test1# Anomaly|Non-Repeatable Read - Isolation|Read Committed", "test2# Anomaly|Non-Repeatable Read - Isolation|Repeatable Read", "test3# Anomaly|Lost Update - Isolation|Read Committed", "test4# Anomaly|Lost Update - Isolation|Repeatable Read", "test5# Anomaly|Ghost Update - Isolation|Read Committed", "test6# Anomaly|Ghost Update - Isolation|Repeatable Read", "test7# Anomaly|Phantom Update - Isolation|Read Committed", "test8# Anomaly|Phantom Update - Isolation|Repeatable Read", "test9# Anomaly|Read Skew - Isolation|Read Committed", "test10# Anomaly|Read Skew - Isolation|Repeatable Read", "test11# Anomaly|Write Skew - Isolation|Read Committed", and "test12# Anomaly|Write Skew - Isolation|Repeatable Read". Below the list are two lines of instructions: "~ Restart mysql ~ (if an error occurs, restart the service and re-run the test)" and "~ EXIT ~".

```
Terminal - dbtech@debian10: ~/dbtech/diamantidis/pycharm/mvcc_remote
File Edit View Terminal Tabs Help

Choose a test to run in mysql dbms:

==> test1# Anomaly|Non-Repeatable Read - Isolation|Read Committed
test2# Anomaly|Non-Repeatable Read - Isolation|Repeatable Read
test3# Anomaly|Lost Update - Isolation|Read Committed
test4# Anomaly|Lost Update - Isolation|Repeatable Read
test5# Anomaly|Ghost Update - Isolation|Read Committed
test6# Anomaly|Ghost Update - Isolation|Repeatable Read
test7# Anomaly|Phantom Update - Isolation|Read Committed
test8# Anomaly|Phantom Update - Isolation|Repeatable Read
test9# Anomaly|Read Skew - Isolation|Read Committed
test10# Anomaly|Read Skew - Isolation|Repeatable Read
test11# Anomaly|Write Skew - Isolation|Read Committed
test12# Anomaly|Write Skew - Isolation|Repeatable Read
~ Restart mysql ~ (if an error occurs, restart the service and re-run the test)
~ EXIT ~
```

Figure A.2: Test scenario selection

These are the predefined test scenarios that a user can choose from. After selecting one, e.g. test2#, the following screen is displayed:

```

MYSQL - TEST2 - # Anomaly|Non-Repeatable Read - Isolation|Repeatable Read
File Edit View Terminal Tabs Help

MariaDB [testdb]>
MariaDB [testdb]> -- S T E P 1 --
MariaDB [testdb]> -----
MariaDB [testdb]> SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> BEGIN;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> SELECT * FROM T WHERE id=1;
+-----+
| id | x |
+-----+
| 1 | 1 |
+-----+
1 row in set (0.000 sec)

MariaDB [testdb]> -- S T E P 3 --
MariaDB [testdb]> -----
MariaDB [testdb]> SELECT * FROM T WHERE id=1;
+-----+
| id | x |
+-----+
| 1 | 1 |
+-----+
1 row in set (0.000 sec)

MariaDB [testdb]> COMMIT;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> SELECT * FROM T ORDER BY id;
+-----+
| id | x |
+-----+
| 1 | 12 |
| 2 | 2 |
| 3 | 3 |
+-----+
3 rows in set (0.000 sec)

MariaDB [testdb]> COMMIT;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]>

MariaDB [testdb]>
MariaDB [testdb]> -- S T E P 2 --
MariaDB [testdb]> -----
MariaDB [testdb]> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> BEGIN;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> UPDATE T SET x=12 WHERE id=1;
Query OK, 1 row affected (0.003 sec)
Rows matched: 1 Changed: 1 Warnings: 0

MariaDB [testdb]> SELECT * FROM T WHERE id=1;
+-----+
| id | x |
+-----+
| 1 | 12 |
+-----+
1 row in set (0.000 sec)

MariaDB [testdb]> COMMIT;
Query OK, 0 rows affected (0.002 sec)

MariaDB [testdb]>

[test2]Swi0: bash- 1: test2|Switch Panes = (CTRL + B) + Arrows. Scroll Mode = (CTRL+ B) + "[". Quit = "q" |*
"debian10" 15:15 14-Jun-20

```

Figure A.3: Test scenario executed

These are two fully interactive shells.

To move the cursor from the left pane to the right, press CTRL + B, release them and then press the Right Arrow key.

To scroll up, "scroll mode" must be enabled. To enable, press CTRL + B, release them and then press "[". To exit "scroll mode", simply press "q".

```

407
408
409 mysql-tests:
410   test1: <5 keys>
437
438   test2:  # Anomaly/Non-Repeatable Read - Isolation/Repeatable Read
439     step1_T1:
440       - '--  S T E P  1  --'
441       - '-----'
442       - 'SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;;'
443       - 'BEGIN;;'
444       - 'SELECT * FROM T WHERE id=1;;'
445     step2_T2:
446       - '--  S T E P  2  --'
447       - '-----'
448       - 'SET TRANSACTION ISOLATION LEVEL READ COMMITTED;;'
449       - 'BEGIN;;'
450       - 'UPDATE T SET x=12 WHERE id=1;;'
451       - 'SELECT * FROM T WHERE id=1;;'
452       - 'COMMIT;;'
453     step3_T1:
454       - '--  S T E P  3  --'
455       - '-----'
456       - 'SELECT * FROM T WHERE id=1;;'
457       - 'COMMIT;;'
458       - 'SELECT * FROM T ORDER BY id;;'
459       - 'COMMIT;;'
460
461   test3: <3 keys>

```

Figure A.4: YAML file with the test#2 steps

Figure A.4 shows how test#2 steps were defined inside the mvcc_tests.yml file. These can be extended, and anyone can add any sequence of SQL steps to be executed.

Each step must be in the format **step**<sequence_number>_T<transaction_number>.

More details can be found inside the mvcc_tests.yml file in the form of comments.