



Assignment Title: **Database Management Systems**

Module Title: **ACCC5001 Database Management Systems**

Lecturer: **Modak Koushik**

Student Number: **1815743**

Student: **Anghel Sandra-Stefania**

Table of Contents

List of Figures	2
Database management system (DBMS) Transaction and Concurrency	1
ACID properties	1
Atomicity.....	1
Consistency.....	3
Isolation.....	4
Durability	5
Concurrency anomalies	5
Dirty Read	5
Unrepeatable Read	6
Lost Update.....	7
Two-phase locking (2PL) protocol.....	8
Security strategy proposal.....	10
Identified problem.....	10
Proposed solution.....	10
Bibliography	16

List of Figures

Figure 1: Transaction - atomicity – SQL Server query example.....	2
Figure 2: Transaction - consistency – SQL Server query example	3
Figure 3: Transaction - isolation – SQL Server query example.....	5
Figure 4: Dirty Read anomaly.....	6
Figure 5: Unrepeatable Read anomaly.....	7
Figure 6: Lost Update anomaly	8
Figure 7: Two-phase locking protocol.....	9
Figure 8: PRGDatabaseAudit table	10
Figure 9: PRGDatabaseAuditLogger trigger.....	10
Figure 10: PRGDatabaseAuditLogger trigger creation SQL query.....	11
Figure 11: PRGDatabaseAuditLogger trigger testing queries	11
Figure 12: Retrieve records from PRGDatabaseAudit table query.....	12
Figure 13: PRGDatabaseAudit table records	12
Figure 14: PRGPropertyAudit table	12
Figure 15: Lease table triggers.....	13
Figure 16: PRGPropertyAuditInsertLogger trigger creation SQL query	13
Figure 17PRGPropertyAuditUpdateLogger trigger creation SQL query.....	13
Figure 18: PRGPropertyAuditDeleteLogger trigger creation SQL query	13
Figure 19: Lease table triggers testing queries.....	14
Figure 20: Retrieve records from PRGPropertyAudit table query	14
Figure 21: PRGPropertyAudit table records	14
Figure 22: PRGPropertyAudit oldValue column XML example	15

Database management system (DBMS) Transaction and Concurrency

A Database Management System is a system software used for performing administrative operations (e.g., performance monitoring, backup and recovery, change management, compliance etc) and offers an interface for users to manage databases [1].

A DBMS is a set of cooperating components including query compiler, execution engine, storage manager, logging/recovery, transaction manager and concurrency control component. The term transaction, in the context of DBMS, refers to a set of logically related operations that are executed on a database [2]. A *transaction* may simply be defined as a logical unit of work. Any transaction contains the following steps:

1. Begin the transaction
2. Execute the set of operations
3. Commit the transaction – if all operations successfully completed
4. Rollback the transaction – if one or more operations failed

Transaction manager is a component of DBMS responsible for creating transactions and managing them. Multiple transactions can be executed at the same time on the same database. The ability to control the simultaneously executions and support multiple users defines the concept of database *concurrency* [2].

ACID properties

In the context of Database Management System transactions, ACID is an acronym representing the main four characteristics of a transaction: *atomicity*, *consistency*, *isolation* and *durability* [3].

Atomicity

Atomicity describes how the operations are executed. All operations (e.g., Data Definition Language (DDL) operations – create, alter, drop; Data Manipulation Language (DML) operations – select, insert, update, delete; Data Control Language (DCL) operations – grant, revoke; Transaction Control Language (TCL) operations – commit, rollback) combined into a transaction must be executed as a single workload. If the workload failed or completed partially, then the transaction must not leave information from the partial executions. The state

of the database at the end of transaction must be the same as it was at the beginning of transaction if any failure occurred [3] [4].

Let us assume the registration of a new property requires two mandatory steps processed by the To-Rent estate agent:

- A new record is inserted into Owner table
- A new record is inserted into Property table

In this case, atomicity involves successful completion of both steps. For example, the atomicity characteristic is violated if the new property record cannot be inserted and the new owner remains inserted at the end of the transaction.

```
USE ToRentDb
GO
SELECT * FROM dbo.[Owner]
BEGIN TRANSACTION atomocity

BEGIN TRY

-- step 1 - A new record is inserted into Owner table

INSERT INTO [dbo].[Owner]([ownerName],[ownerAddress]
,[ownerPostcode],[telephoneNumber],[owneremail])
VALUES
('Sandra Anghel'
,'London'
,'E15 3NN'
,'07943652218'
,'sandra.anghel@outlook.com')

-- step 2 - A new record is inserted into Property table

INSERT INTO [dbo].[Property]([propertyAddress],[propertyCityID]
,[propertyPostcode],[propertyType]
,[noofrooms],[monthlyRent],[ownerID])
VALUES
('Bettons Park'
,1
,'E15 4NN'
,'studio'
,1
,850
,999) ---- ownerId = 999 is not existing.

COMMIT TRANSACTION atomocity

END TRY

BEGIN CATCH
rollback TRANSACTION atomocity
END CATCH
```

	ownerID	ownerName	ownerAddress	ownerPostcode	telephoneNumber	owneremail
1	1	Adam Bailey	61 Wellfield Road	L30 4GB	07935263223	adam.bailey@gmail.com
2	2	Mauglen Camber	91 Western Road	NE1 4ST	07824563223	mauglen.camber@gmail.com
3	3	Rebecca Ferrye	44-46 Morningside Road	S2 4SU	07994563227	rebecca.ferrye@gmail.com
4	4	Alexander Baites	27 Colmore Row	SN38 1NW	07934565223	alexander.baites@gmail.com
5	5	Christofer Gucyne	39 Acacia Avenue	E14 5HQ	07930953723	christofer.gucyne@gmail.com
6	6	Gabriell Crage	501a Halfway Street	BX2 1LB	07924563003	gabriell.crage@gmail.com
7	7	Jarret Ludge	10 Downing Street	M50 2QH	07974500223	jarret.ludge@gmail.com

	ownerID	ownerName	ownerAddress	ownerPostcode	telephoneNumber	owneremail
1	1	Adam Bailey	61 Wellfield Road	L30 4GB	07935263223	adam.bailey@gmail.com
2	2	Mauglen Camber	91 Western Road	NE1 4ST	07824563223	mauglen.camber@gmail.com
3	3	Rebecca Ferrye	44-46 Morningside Road	S2 4SU	07994563227	rebecca.ferrye@gmail.com
4	4	Alexander Baites	27 Colmore Row	SN38 1NW	07934565223	alexander.baites@gmail.com
5	5	Christofer Gucyne	39 Acacia Avenue	E14 5HQ	07930953723	christofer.gucyne@gmail.com
6	6	Gabriell Crage	501a Halfway Street	BX2 1LB	07924563003	gabriell.crage@gmail.com
7	7	Jarret Ludge	10 Downing Street	M50 2QH	07974500223	jarret.ludge@gmail.com

Figure 1: Transaction - atomicity – SQL Server query example

Try-catch mechanism is used to ensure the atomicity of the transaction. In case of an error inside the try block then a rollback is executed. If all the steps are successfully executed, then the transaction is committed. In figure 1, because owner id 999 does not exist, then the second step will not be successfully executed, and the transaction is rolled back. Owner ‘Sandra Anghel’ is not persisted into the database.

Consistency

Consistency is a property that refers to not violating the data integrity constraints. If the database is not entered into an illegal state, a transaction is consistent, and it must satisfy columns types, primary keys, foreign keys or other constraints. Users are responsible for ensuring consistency property of a transaction [3] [4].

Some of the fraudulent activities of To-Rent estate agent could result in an inconsistent state of the database if the foreign key associated with the existing clientId column in the Lease table is bypassed and a new/or existing leased property is assigned to an inexistent client.

```
USE ToRentDb
GO
SELECT * FROM dbo.[Lease]
BEGIN TRANSACTION consistency

BEGIN TRY

    UPDATE dbo.[Lease]
    SET clientId = 999 ---- clientId = 999 is not existing.
    WHERE leaseNumber = 1

    COMMIT TRANSACTION consistency

END TRY

BEGIN CATCH
    ROLLBACK TRANSACTION consistency
END CATCH

SELECT * FROM dbo.[Lease]
```

	leaseNumber	clientId	ownerID	propertyID	methodofpay	depositPaidStatus	startDate	endDate
1	1	3	1	1	cash	0	2020-05-01 00:00:00.000	2021-05-01 00:00:00.000
2	2	5	2	2	card	1	2020-07-05 00:00:00.000	2021-07-05 00:00:00.000
3	3	7	4	4	cash	1	2020-09-20 00:00:00.000	2022-09-20 00:00:00.000
4	4	9	6	6	card	0	2020-11-30 00:00:00.000	2021-05-30 00:00:00.000
5	11	5	2	5	cash	0	2020-11-26 19:42:22.887	2020-11-26 19:42:22.887

	leaseNumber	clientId	ownerID	propertyID	methodofpay	depositPaidStatus	startDate	endDate
1	1	3	1	1	cash	0	2020-05-01 00:00:00.000	2021-05-01 00:00:00.000
2	2	5	2	2	card	1	2020-07-05 00:00:00.000	2021-07-05 00:00:00.000
3	3	7	4	4	cash	1	2020-09-20 00:00:00.000	2022-09-20 00:00:00.000
4	4	9	6	6	card	0	2020-11-30 00:00:00.000	2021-05-30 00:00:00.000
5	11	5	2	5	cash	0	2020-11-26 19:42:22.887	2020-11-26 19:42:22.887

Figure 2: Transaction - consistency – SQL Server query example

Foreign key constraint is checked, and the transaction did not change the value of ClientID property.

Isolation

Isolation describes the behaviour of a transaction reported to other transactions' execution. If multiple transactions are running in parallel, the operations happening in each of them must be isolated. One transaction must not expose its events to another transaction during execution and it must not be affected by other transactions [3].

If the To-Rent estate agent executes two transactions: T1 – register a new property and T2 – lease and existing property, then the operations executed in T1 must not affect the operations executed in T2 and the transactions must be isolated. Isolation level can be set into transactions [5]:

```
USE ToRentDb;
GO
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
GO
BEGIN TRANSACTION;
GO
SELECT top 1 * FROM dbo.Property;
SELECT * FROM sys.dm_tran_locks WHERE resource_associated_entity_id = object_id('dbo.Property')
GO
SELECT top 1 * FROM dbo.Lease;
SELECT * FROM sys.dm_tran_locks WHERE resource_associated_entity_id = object_id('dbo.Lease')
GO
COMMIT TRANSACTION;
SELECT * FROM sys.dm_tran_locks WHERE resource_associated_entity_id = object_id('dbo.Property')
SELECT * FROM sys.dm_tran_locks WHERE resource_associated_entity_id = object_id('dbo.Lease')
GO
```

1	propertyID	propertyAddress	propertyCityID	propertyPostcode	propertyType	noofrooms	monthlyRent	ownerID	Select property table	
1	1	33 Norton Road	1	BN1 1AA	flat	1	1000	1		
1	resource_type	resource_subtype	resource_database_id	resource_description	resource_associated_entity_id	resource_lock_partition	request_mode	request_type	request_status	
1	OBJECT		9		450100644	0	IS	LOCK	GRANT	
property table - locked										
<										
1	leaseNumber	clientID	ownerID	propertyID	methodofpay	depositPaidStatus	startDate	endDate	Select lease table	
1	1	3	1	1	cash	0	2020-05-01 00:00:00.000	2021-05-01 00:00:00.000		
1	resource_type	resource_subtype	resource_database_id	resource_description	resource_associated_entity_id	resource_lock_partition	request_mode	request_type	request_status	
1	OBJECT		9		1157579162	0	IS	LOCK	GRANT	
lease table - locked										
<										
property table - lock released (after transaction committed)										
<										
lease table - lock released (after transaction committed)										

Figure 3: Transaction - isolation – SQL Server query example

Isolation level is set at the beginning of the transaction. Once the transaction is committed then the locks are released.

Durability

Durability is a transaction property that refers to the persistency of data once the transaction is committed. After committing, even if the system fails or a restart is carried out, data must not be compromised [4].

Concurrency anomalies

Transaction can be executed in parallel. The fact that same data can be modified from multiple concurrent transactions between read or write operations leads to different problems called *concurrency anomalies*. Three of the main concurrency anomalies are: dirty read, unrepeatable read and lost update [2].

Dirty Read

Dirty Read anomaly occurs when a transaction can read data that is modified and not yet committed by another transaction that is running in parallel [6].

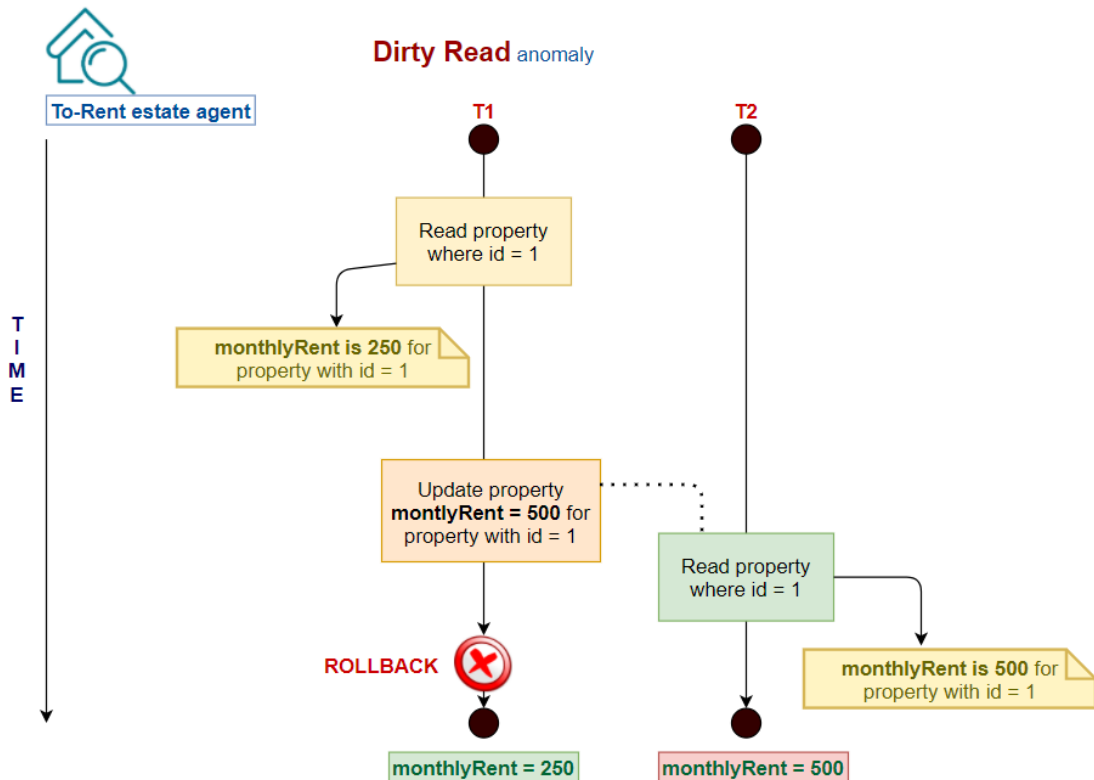


Figure 4: Dirty Read anomaly

Suppose that transaction T1 is responsible for updating the monthly rent for a property based on a notification email received from the owner of the property and transaction T2 is responsible for sending the details of the property to a client interested in renting it.

Transaction T1 executes a read operation of the details about the property with id 1 from the Property table. At this moment, the property's monthlyRent is 250. Later, the monthlyRent column is updated to 500. Before committing the transaction T1, transaction T2 reads the property with id 1 details and results a monthlyRent equals with 500. Transaction T1 cannot commit the changes and a rollback is performed, but transaction T2 is not notified so it uses the wrong value for monthly rent. In this case, a dirty read anomaly occurs, and wrong data is passed to the client.

Unrepeatable Read

Unrepeatable Read anomaly occurs when during execution a transaction reads same row twice and its value is different each time [6].

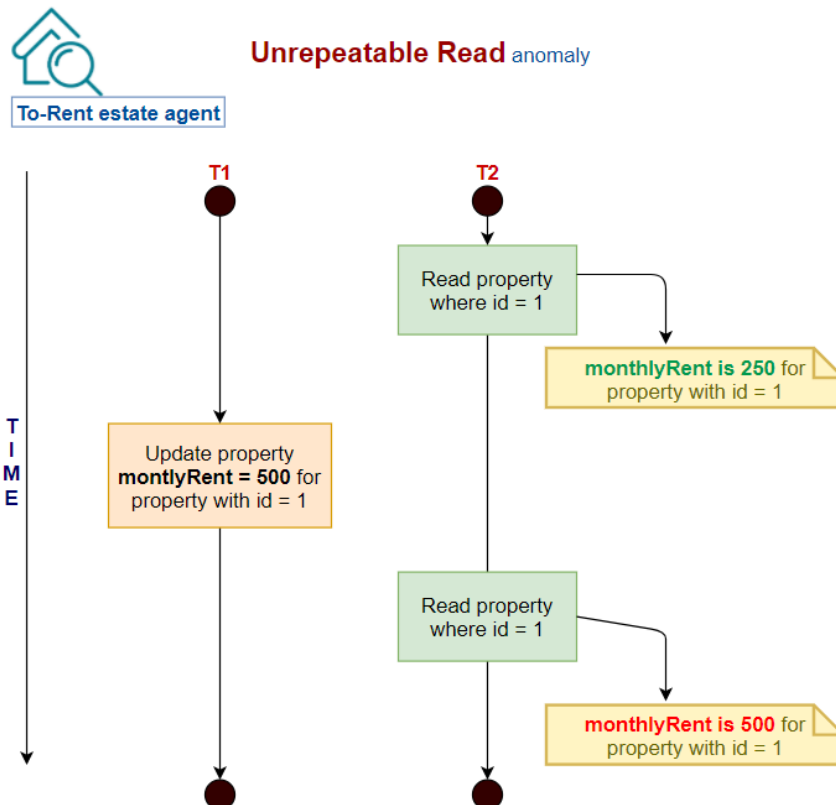


Figure 5: Unrepeatable Read anomaly

Suppose that transaction T1 is responsible for updating the monthly rent for a property based on a notification email received from the owner of the property and transaction T2 is responsible for generating a property flyer and sending the details of the property to a client interested in renting it.

Transaction T2 perform two read operations of the same property row existing into Property tables. Based on this scenario, the flyer will display monthly rent equals with 250 and the email for the client will contains monthly rent equals with 500, because the monthly rent for property with id 1 was updated by transaction T1.

Lost Update

Lost Update anomaly occurs when two transactions interleaves their operations in a way that makes the row incorrect [6].

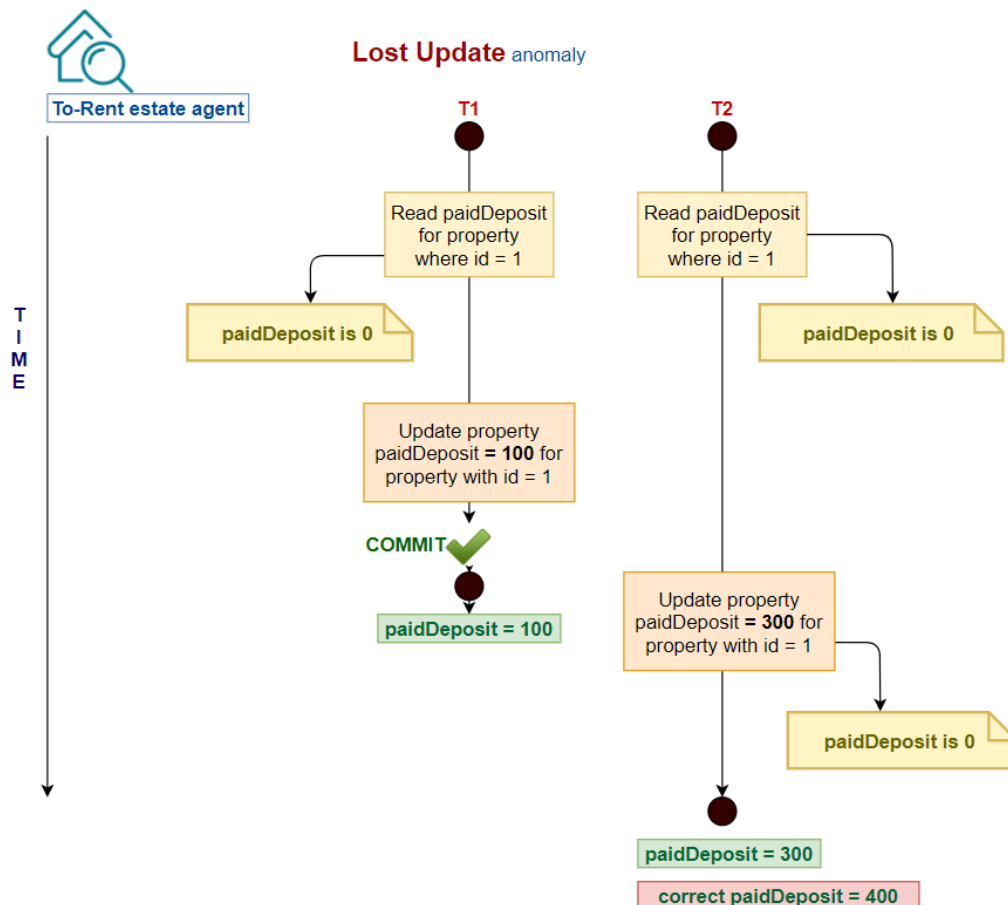


Figure 6: Lost Update anomaly

Suppose that transaction T1 is responsible for updating the deposit paid when the client visited the property and transaction T2 is responsible for updating the new paid deposit amount. When the client made the first visit, the deposit amount was not registered into the database.

The paid deposit amount was read at the beginning of both transactions. As transaction T1 updated and committed the value 100 and another read operation was not performed in transaction T2 then the final deposit value is 300 instead of 400 (T1 paidDeposit 100 + T2 paidDeposit 300).

Two-phase locking (2PL) protocol

Database use locks to protect resources (connections, table, data rows etc.) that are shared. There are two types of locks: read lock and write lock. A read lock does not allow writing operating on the locked resource. A write lock does not allow both read and write operations on a locked resource. Two-phase locking protocol is a mechanism used for ensuring data integrity. It splits the transaction into two phases [6]:

- Growing phase – the transaction can acquire locks but cannot release locks

- Shrinking phase – the transaction can release locks but cannot acquire locks

Based on previous dirty read scenario, two-phase locking protocol can be used as following:

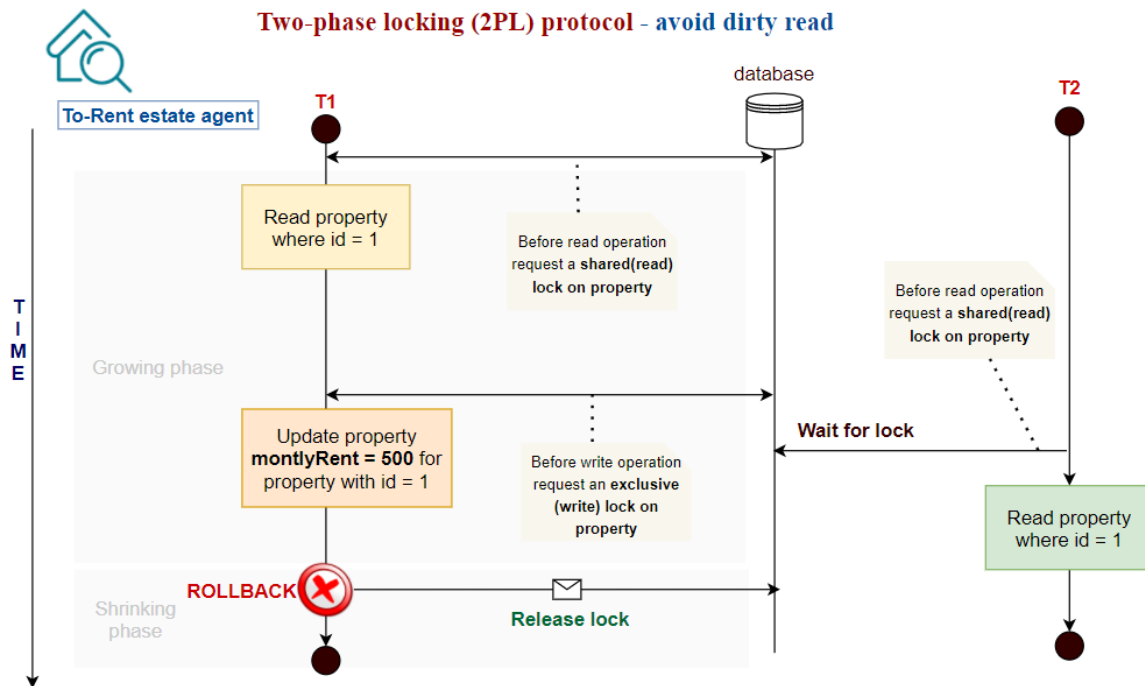


Figure 7: Two-phase locking protocol

Transaction T1 acquires a shared (read) lock on product resource. In this was the resource can be read but it cannot be changed. The next operation executed by transaction T1 represents an update. Before executing it, the transaction acquires an exclusive (write) lock on the property resource. Transaction T2 requires a shred lock on the same resource in order to read property details. But because the lock cannot be granted, as it already exists, transaction T2 is suspended until Transaction T1 rollbacks and releases the lock. Once the lock is release transaction T2 can grant its own requested lock and the value read is valid.

Security strategy proposal

Identified problem

To-Rent is an estate agent that rapidly expands their business to different regions in the UK. The management identified fraudulent activities such as register fake visits into the system and assigned leased properties to them. The following activities are suspected:

- Tables are created in order to hide properties and client details
- Leased properties are deleted from the Property table

Proposed solution

The suggested solution requires the creation of database triggers in order to stop these activities and discover the responsible persons.

First illegal activity refers to DDL operations - *tables are created in order to hide properties and client details*. A new table named *PRGDatabaseAudit* was created to store details (database name, table name, event type, login name, SQL command, audit date and time) about the *create*, *alter* and *drop* table operations.

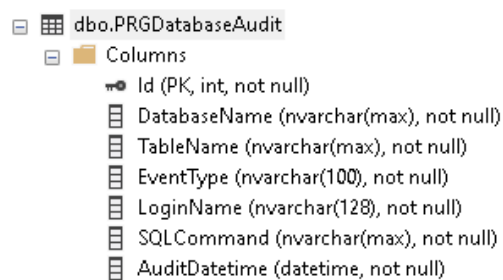


Figure 8: PRGDatabaseAudit table

A new trigger named *PRGDatabaseAuditLogger* was created to insert records into *PRGDatabaseAudit* table.

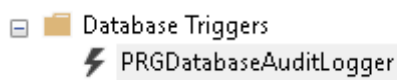


Figure 9: PRGDatabaseAuditLogger trigger

```

CREATE TRIGGER [PRGDatabaseAuditLogger]
ON DATABASE
FOR CREATE_TABLE, ALTER_TABLE, DROP_TABLE
AS

DECLARE @eventData XML
SET @eventData = EVENTDATA()

INSERT INTO PRGDatabaseAudit (DatabaseName, TableName, EventType, LoginName, SQLCommand, AuditDatetime)
VALUES (
    @eventData.value('(/EVENT_INSTANCE/DatabaseName)[1]', 'nvarchar(max)'),
    @eventData.value('(/EVENT_INSTANCE/ObjectName)[1]', 'nvarchar(max)'),
    @eventData.value('(/EVENT_INSTANCE/EventType)[1]', 'nvarchar(100)'),
    @eventData.value('(/EVENT_INSTANCE/LoginName)[1]', 'nvarchar(128)'),
    @eventData.value('(/EVENT_INSTANCE/TSQLCommand/CommandText)[1]', 'nvarchar(max)'),
    @eventData.value('(/EVENT_INSTANCE/PostTime)[1]', 'datetime')
)

GO

ENABLE TRIGGER [PRGDatabaseAuditLogger] ON DATABASE
GO

```

Figure 10: PRGDatabaseAuditLogger trigger creation SQL query

This trigger will capture any illegal event related to DDL commands (create, alter or drop). To-Rent administrator can run the following queries in order to test this first step of the security plan.

```

USE [ToRentDb]
GO

--Test PRGDatabaseAuditLogger trigger

-- test Create event
CREATE TABLE PRGDatabaseAuditLogger_TestTable (id INT);
GO
SELECT * FROM PRGDatabaseAudit;
GO

-- test Alter event
ALTER TABLE PRGDatabaseAuditLogger_TestTable
ADD title nvarchar(100)
GO
SELECT * FROM PRGDatabaseAudit;
GO

-- test Drop event
DROP TABLE PRGDatabaseAuditLogger_TestTable
GO
SELECT * FROM PRGDatabaseAudit;
GO

```

Figure 11: PRGDatabaseAuditLogger trigger testing queries

The records will be inserted into *PRGDatabaseAudit* table and a select SQL command retrieves them.

```

| SELECT [Id]
      ,[DatabaseName]
      ,[TableName]
      ,[EventType]
      ,[LoginName]
      ,[SQLCommand]
      ,[AuditDatetime]
FROM [ToRentDb].[dbo].[PRGDatabaseAudit]

```

Figure 12: Retrieve records from PRGDatabaseAudit table query

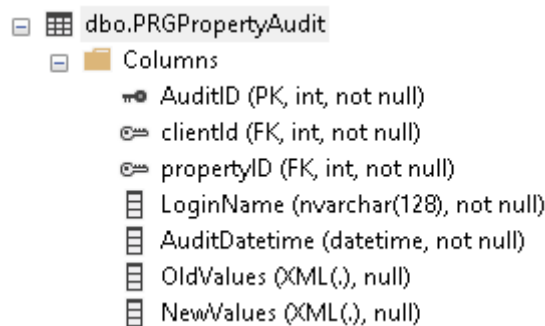
Id	DatabaseName	TableName	EventType	LoginName
1	ToRentDb	PRGDatabaseAuditLogger_TestTable	CREATE_TABLE	MicrosoftAccount\sandra.anghel2701@gmail.com
2	ToRentDb	PRGDatabaseAuditLogger_TestTable	ALTER_TABLE	MicrosoftAccount\sandra.anghel2701@gmail.com
3	ToRentDb	PRGDatabaseAuditLogger_TestTable	DROP_TABLE	MicrosoftAccount\sandra.anghel2701@gmail.com

SQLCommand	AuditDatetime
CREATE TABLE PRGDatabaseAuditLogger_TestTable (id INT)	2020-11-26 18:43:31.700
ALTER TABLE PRGDatabaseAuditLogger_TestTable ADD title nvarchar(100)	2020-11-26 18:43:31.800
DROP TABLE PRGDatabaseAuditLogger_TestTable	2020-11-26 18:43:31.870

Figure 13: PRGDatabaseAudit table records

The records indicated what action was executed (EventType), who is the responsible person (LoginName), what table object is the result of the event (TableName), and when the event occurred.

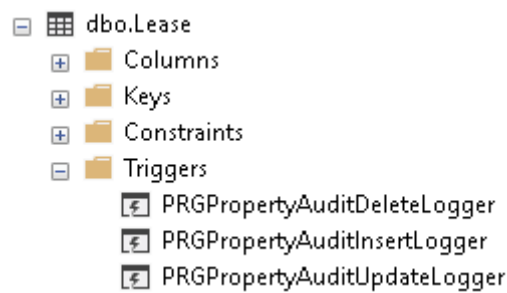
The second illegal activity refers to DML operations - *leased properties are deleted from the Property table*. A new table named *PRGPropertyAudit* was created to store details (a unique identifier for the audit record and property information) about the *insert*, *update* and *delete* operations.



dbo.PRGPropertyAudit
Columns
AuditID (PK, int, not null)
clientID (FK, int, not null)
propertyID (FK, int, not null)
LoginName (nvarchar(128), not null)
AuditDatetime (datetime, not null)
OldValues (XML(), null)
NewValues (XML(), null)

Figure 14: PRGPropertyAudit table

Three new triggers *PRGPropertyAuditDeleteLogger*, *PRGPropertyAuditInsertLogger*, *PRGPropertyAuditUpdateLogger* one for each operation, were created on Lease table to insert records into *PRGPropertyAudit* table.



dbo.Lease
Columns
Keys
Constraints
Triggers
PRGPropertyAuditDeleteLogger
PRGPropertyAuditInsertLogger
PRGPropertyAuditUpdateLogger

Figure 15: Lease table triggers

```
CREATE TRIGGER [dbo].[PRGPropertyAuditInsertLogger]
ON [dbo].[Lease]
AFTER INSERT AS

    DECLARE @newValues XML
    SET @newValues = (SELECT * FROM inserted FOR XML PATH('lease'))

    INSERT INTO PRGPropertyAudit(clientId, propertyID, LoginName, AuditDatetime, OldValues, NewValues)
    SELECT i.clientID, i.propertyID, ORIGINAL_LOGIN(), GETDATE(), null, @newValues
    FROM inserted i

GO

ALTER TABLE [dbo].[Lease] ENABLE TRIGGER [PRGPropertyAuditInsertLogger]
GO
```

Figure 16: PRGPropertyAuditInsertLogger trigger creation SQL query

```
CREATE TRIGGER [dbo].[PRGPropertyAuditUpdateLogger]
ON [dbo].[Lease]
AFTER UPDATE AS

    DECLARE @oldValues XML
    SET @oldValues = (SELECT * FROM deleted FOR XML PATH('lease'))

    DECLARE @newValues XML
    SET @newValues = (SELECT * FROM inserted FOR XML PATH('lease'))

    INSERT INTO PRGPropertyAudit(clientId, propertyID, LoginName, AuditDatetime, OldValues, NewValues)
    SELECT i.clientID, i.propertyID, ORIGINAL_LOGIN(), GETDATE(), @oldValues, @newValues
    FROM inserted i

GO

ALTER TABLE [dbo].[Lease] ENABLE TRIGGER [PRGPropertyAuditUpdateLogger]
GO
```

Figure 17 PRGPropertyAuditUpdateLogger trigger creation SQL query

```
CREATE TRIGGER [dbo].[PRGPropertyAuditDeleteLogger]
ON [dbo].[Lease]
AFTER DELETE AS

    DECLARE @oldValues XML
    SET @oldValues = (SELECT * FROM deleted FOR XML PATH('lease'))

    INSERT INTO PRGPropertyAudit(clientId, propertyID, LoginName, AuditDatetime, OldValues, NewValues)
    SELECT d.clientID, d.propertyID, ORIGINAL_LOGIN(), GETDATE(), @oldValues, NULL
    FROM deleted d

GO

ALTER TABLE [dbo].[Lease] ENABLE TRIGGER [PRGPropertyAuditDeleteLogger]
GO
```

Figure 18: PRGPropertyAuditDeleteLogger trigger creation SQL query

This trigger will capture any illegal event related to DML commands (insert, update or delete) on Lease table. To-Rent administrator can run the following queries in order to test the second step of the security plan.

```

USE [ToRentDb]
GO

DELETE FROM [dbo].[PRGPropertyAudit]

--Test PRGPropertyAuditInsertLogger trigger

INSERT INTO [dbo].[Lease]
    ([leaseNumber]
    ,[clientID]
    ,[ownerID]
    ,[propertyID]
    ,[methodofpay]
    ,[depositPaidStatus]
    ,[startDate]
    ,[endDate])
VALUES
    (88
    ,1
    ,1
    ,1
    ,'cash'
    ,0
    ,GETDATE()
    ,GETDATE()+1)

--Test PRGPropertyAuditUpdateLogger trigger

UPDATE [dbo].[Lease]
SET [clientID] = 2
WHERE leaseNumber = 88

UPDATE [dbo].[Lease]
SET [ownerID] = 3
WHERE leaseNumber = 88

UPDATE [dbo].[Lease]
SET [propertyID] = 4
WHERE leaseNumber = 88

UPDATE [dbo].[Lease]
SET [methodofpay] = 'card'
WHERE leaseNumber = 88

UPDATE [dbo].[Lease]
SET [depositPaidStatus] = 1
WHERE leaseNumber = 88

UPDATE [dbo].[Lease]
SET [startDate] = GETDATE() + 5
WHERE leaseNumber = 88

UPDATE [dbo].[Lease]
SET [endDate] = GETDATE() + 230
WHERE leaseNumber = 88

--Test PRGPropertyAuditDeleteLogger trigger

DELETE FROM [dbo].[Lease]
WHERE leaseNumber = 88

```

Figure 19: Lease table triggers testing queries

The records will be inserted into *PRGPropertyAudit* table and a select SQL command retrieves them.

```

SELECT [AuditID]
,[clientID]
,[propertyID]
,[LoginName]
,[AuditDatetime]
,[OldValues]
,[NewValues]
FROM [ToRentDb].[dbo].[PRGPropertyAudit]

```

Figure 20: Retrieve records from *PRGPropertyAudit* table query

AuditID	clientID	propertyID	LoginName	AuditDatetime	OldValues	NewValues
12	1	1	MicrosoftAccount\sandra.anghel2701@gmail.com	2020-11-26 19:55:05.073	NULL	<lease><leaseNumber>88</leaseNumber><clientID>1</clientID>
13	2	1	MicrosoftAccount\sandra.anghel2701@gmail.com	2020-11-26 19:55:05.077	<lease><leaseNumber>88</leaseNumber><clientID>1</clientID>	<lease><leaseNumber>88</leaseNumber><clientID>2</clientID>
14	2	1	MicrosoftAccount\sandra.anghel2701@gmail.com	2020-11-26 19:55:05.077	<lease><leaseNumber>88</leaseNumber><clientID>2</clientID>	<lease><leaseNumber>88</leaseNumber><clientID>2</clientID>
15	2	4	MicrosoftAccount\sandra.anghel2701@gmail.com	2020-11-26 19:55:05.077	<lease><leaseNumber>88</leaseNumber><clientID>2</clientID>	<lease><leaseNumber>88</leaseNumber><clientID>2</clientID>
16	2	4	MicrosoftAccount\sandra.anghel2701@gmail.com	2020-11-26 19:55:05.080	<lease><leaseNumber>88</leaseNumber><clientID>2</clientID>	<lease><leaseNumber>88</leaseNumber><clientID>2</clientID>
17	2	4	MicrosoftAccount\sandra.anghel2701@gmail.com	2020-11-26 19:55:05.080	<lease><leaseNumber>88</leaseNumber><clientID>2</clientID>	<lease><leaseNumber>88</leaseNumber><clientID>2</clientID>
18	2	4	MicrosoftAccount\sandra.anghel2701@gmail.com	2020-11-26 19:55:05.080	<lease><leaseNumber>88</leaseNumber><clientID>2</clientID>	<lease><leaseNumber>88</leaseNumber><clientID>2</clientID>
19	2	4	MicrosoftAccount\sandra.anghel2701@gmail.com	2020-11-26 19:55:05.080	<lease><leaseNumber>88</leaseNumber><clientID>2</clientID>	<lease><leaseNumber>88</leaseNumber><clientID>2</clientID>
20	2	4	MicrosoftAccount\sandra.anghel2701@gmail.com	2020-11-26 19:55:05.083	<lease><leaseNumber>88</leaseNumber><clientID>2</clientID>	NULL

Figure 21: *PRGPropertyAudit* table records

The records indicate what action was executed

- Insert operation = oldValues is null and newValues is not null
- Update operation = oldValues is not null and newValues is not null

- Delete operation = oldValues is not null and newValues is null

They also indicate who is the responsible person (LoginName), and when the event occurred and what are the differences. OldValues and NewValues columns store XML (Extensible Markup Language) values.

```
<lease>
  <leaseNumber>88</leaseNumber>
  <clientID>1</clientID>
  <ownerID>1</ownerID>
  <propertyID>1</propertyID>
  <methodofpay>cash</methodofpay>
  <depositPaidStatus>0</depositPaidStatus>
  <startDate>2020-11-26T19:55:05.070</startDate>
  <endDate>2020-11-27T19:55:05.070</endDate>
</lease>
```

Figure 22: PRGPropertyAudit oldValue column XML example

To-Rent administrator can easily identify which property was deleted from Lease table and what update was performed by comparing the old and new xml values.

The usage of triggers is an effective way for To-Rent management to detect potential illegal activities more easily. On the other hand, it can lead to performance issues on long term. The audit tables size will increase over time and it will be difficult to analyse the records.

A better security strategy may include the following recommendations:

- Only database administrators can have direct access to the database
- Each authorized user must have its own credentials
- The system must not perform a hard-delete. A *soft delete* must be implemented
- The system must have a robust logging and auditing strategy (e.g., audit and logging records longer than 30days to be migrated into history tables or even different databases to avoid overloading of the current system database)
- A backup strategy must be defined

Given that these recommendations take time and effort, providing an efficient and fast solution to stop the illegal activities is the best way to satisfy the requirements of the To-Rent estate agent.

Bibliography

- [1] M. L. Gillenson, Fundamentals of Database Management Systems, John Wiley & Sons, 2004.
- [2] D. Korotkevitch, Expert SQL Server Transactions and Locking, Apress, 2018.
- [3] J. Celko, SQL for Smarties, 5th Edition, Morgan Kaufmann, 2014.
- [4] G. Blokdik, ACID Transactions, 2nd Edition, 5STARCook, 2018.
- [5] Microsoft, "SET TRANSACTION ISOLATION LEVEL (Transact-SQL)," 22 10 2018. [Online].
Available: <https://docs.microsoft.com/en-us/sql/t-sql/statements/set-transaction-isolation-level-transact-sql?view=sql-server-ver15>.
- [6] V. Mihalcea, Transactions and Concurrency Control, Cluj-Napoca: Mihalcea Vlad, 2020.