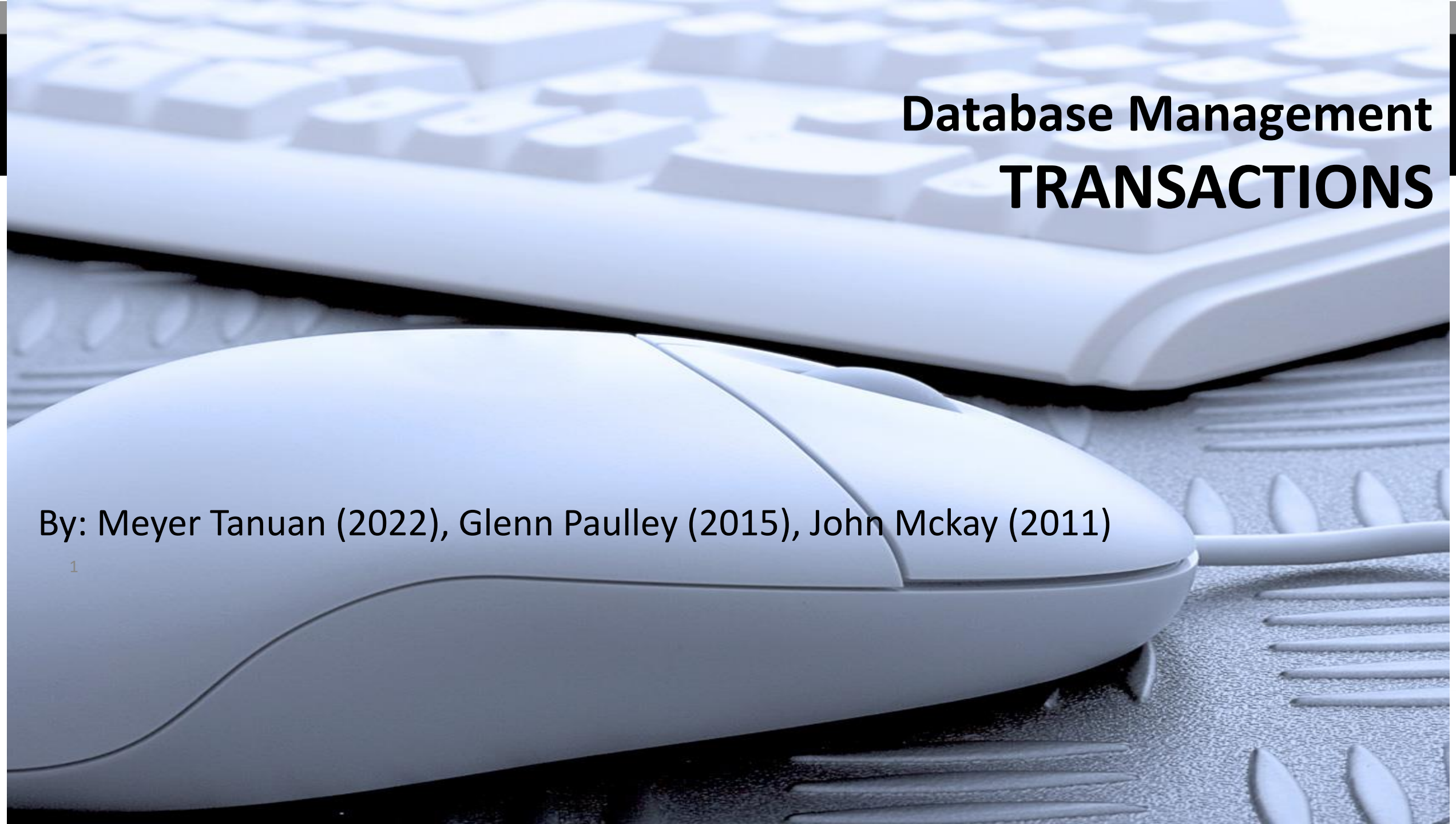


Database Management
TRANSACTIONS

By: Meyer Tanuan (2022), Glenn Paulley (2015), John Mckay (2011)

Transactions and transaction modes

- A transaction is a single logical unit of work
- Basic idea: a transaction either executes in its entirety, or its partial actions are undone
- In practice, a transaction consists of one or more SQL Data Manipulation Language (DML) statements that are logically related
  - INSERT, UPDATE, DELETE, MERGE – all of these statements are individually atomic statements

Transactions and ACID

- BEGIN TRANSACTION
  - Represents the start of a logical unit of work
  - **All** of actions of a transaction must succeed, or the **entire** effect of the transaction must be undone – termed **atomicity** (**A**)
  - A transaction changes the state of the database from one state to another – that state must be logically **consistent** (**C**)
- COMMIT
  - Permanently save the effects of the transaction and makes these changes **durable** (**D**)
- ROLLBACK
  - Discards the state changes of a transaction

Transactions and ACID

- The purpose of transaction processing is to enable users to see a consistent view of database
- Moreover, we would like to be able to write applications assuming that each transaction executes in **isolation** (I) of all of the other concurrently executing transactions
- Otherwise, our model of the system breaks down
- ACID is an acronym for:
  - Atomicity, Consistency, Isolation, Durability

ACID: Atomicity

- Basic idea: transactions are indivisible
- Either all of the SQL statements in a transaction complete (commit), or none of the SQL statements complete (rollback)
- Atomicity also applies to individual SQL statements
  - Enforced automatically by the database management system
  - Either an UPDATE, DELETE, INSERT, or MERGE statement successfully acts on all of the intended rows, or the effect of the statement is undone

ACID: Atomicity Options

BEGIN TRANSACTION

sql statement

sql statement

sql statement

sql statement

COMMIT

BEGIN TRANSACTION

sql statement

sql statement

sql statement

sql statement

ROLLBACK

ACID: Consistency

- Whether a transaction commit or rolls back, the database must be left in a consistent state
- In the case of commit, everything that should happen must happen
- In the case of rollback, the database should be as if the transaction was never attempted.

ACID: Isolation

- Each transaction must be isolated from other concurrent database activity
- Changes made by a transaction should not become visible to other transactions until the transaction commits
- We rely on isolation in order to write applications
- We count on isolation so that we do not need to consider (at least most of the time) the potential impact of other concurrently executing programs

ACID: Durability

- Once a transaction commits, all changes must be guaranteed to be permanent
- If the database crashes, committed transactions must be recovered
  - This is usually accomplished using a database system's forward transaction log
  - Concept: Write-ahead logging (WAL)

Note: if the transaction was NOT committed, it is NOT recovered.

SQL Server transaction modes

1) Explicit transactions
   - Require a **visible** BEGIN TRANSACTION
2) Implicit transactions
   - Equivalent to an **unseen** BEGIN TRANSACTION
   - Continue until COMMIT or ROLLBACK is issued
   - SET IMPLICIT_TRANSACTIONS **ON**
3) Autocommit transactions
   - Bound by an **unseen** BEGIN TRANSACTION and an **unseen** COMMIT TRANSACTION (automatic)
   - Cannot issue a COMMIT or ROLLBACK statement in autocommit mode
   - SET IMPLICIT_TRANSACTIONS **OFF**

1) Explicit Transactions

- Explicit transaction management starts with a **visible** BEGIN TRANSACTION (or BEGIN TRAN)
- Continue until a COMMIT or ROLLBACK
- It is typically done:
  - In a program
  - In a stored procedure

2) Implicit Transactions

- Each SQL DML statement implicitly begins a transaction
- Unless you explicitly COMMIT, the next time a SQL DML statement is processed, a new transaction begins
  - You must still do an explicit rollback to roll back the transaction
- To start using implicit transactions:

  SET IMPLICIT_TRANSACTIONS ON

3) Autocommit Transactions

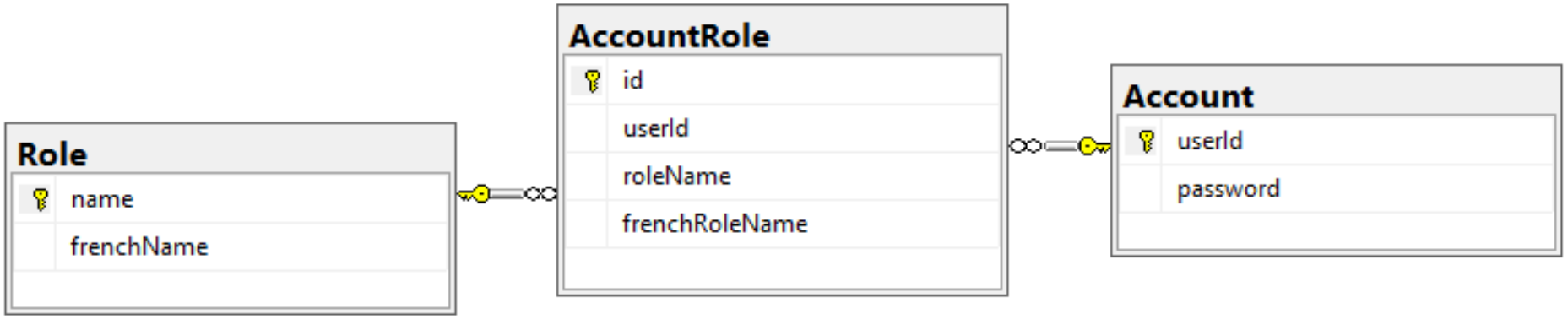
- Autocommit mode is the default transaction management mode of SQL Server.
  - Default: SET IMPLICIT_TRANSACTIONS **OFF**
- Every Transact-SQL statement is committed or rolled back when it completes.
  - If a statement completes successfully, it is committed
  - If it encounters any error, it is rolled back

SQL Server and Autocommit

- A Microsoft SQL Server connection operates in autocommit mode whenever this default mode has not been overridden by either explicit or implicit transactions
- AUTOCOMMIT not only provides poor semantics, it is a performance killer since each COMMIT forces a write of the transaction log to stable storage (disk)
- Autocommit mode is also the default mode for
  - ADO, ADO.NET, OLE DB, ODBC, [Java JDBC,] and DB-Library [ESQL]

Example 1: Role, Account, AccountRole tables

- Add three rows in one transaction:
  - New Role: Teacher (Professeur)
  - New Account: mtanuan (generated password)
  - New AccountRole: mtanuan as Teacher



Step 1: Declare variables and clean up SIS tables

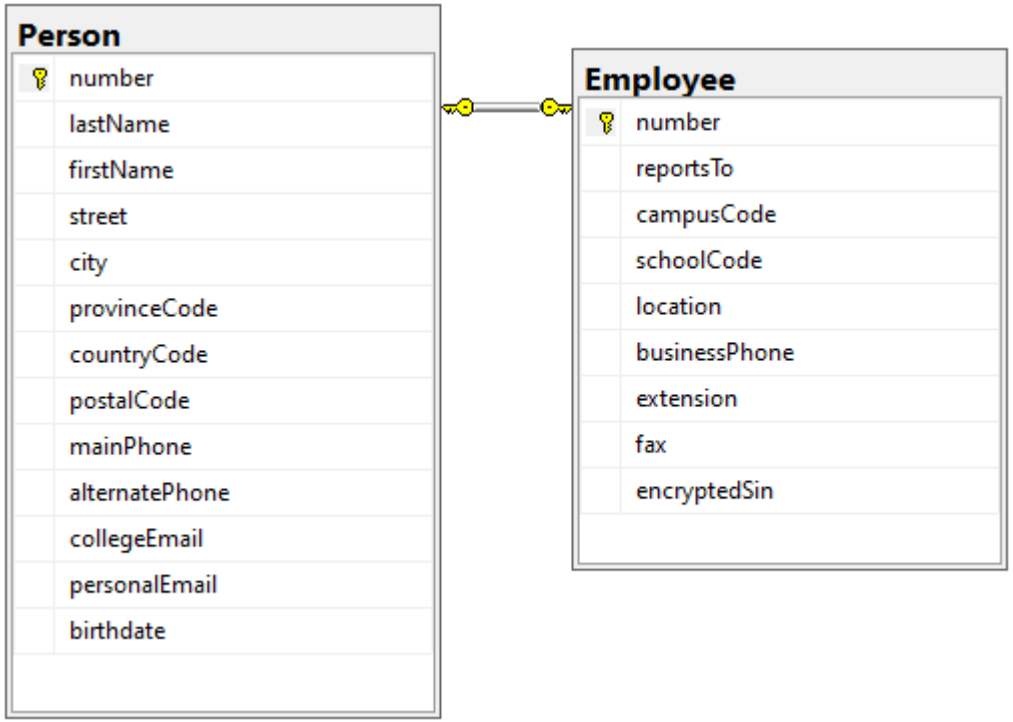
```
DECLARE @UserId NVARCHAR(20) = 'mtanuan';
DECLARE @RoleName NVARCHAR(30) = 'Teacher';
DECLARE @FrenchRoleName NVARCHAR(30) = 'Professeur';

DELETE FROM AccountRole
WHERE userId = @UserId;

DELETE FROM Account
WHERE userId = @UserId;

DELETE FROM Role
WHERE name = @RoleName;
```

Step 2: Insert all records (explicit transaction)

```
BEGIN TRAN;
    INSERT INTO Account
        (userId, password)
    VALUES (@UserId, NEWID());
    INSERT INTO Role
        (name, frenchName)
    VALUES (@RoleName, @FrenchRoleName);
    INSERT INTO AccountRole
        (userId, roleName, frenchRoleName)
    VALUES (@UserId, @RoleName, @FrenchRoleName);
    Print 'New ID created for AccountRole table: ';
    Print @@IDENTITY;
COMMIT TRAN;
```

Example 2: Person and Employee tables

- Add 2 rows in one explicit transaction:
  - New Person: Meyer Tanuan (#ABC1234)
  - New Employee: (#ABC1234) in BUS(iness), W(aterloo), 3G24
- Update an Employee record with ROLLBACK (implicit transaction)



CONESTOGA
Connect Life and Learning

Step 1: Duplicate tables and declare variabes

```
DROP TABLE Person2;
SELECT *   INTO Person2 FROM Person;
TRUNCATE TABLE Person2;
DROP TABLE Employee2;
SELECT * INTO Employee2 FROM Employee;
TRUNCATE TABLE Employee2;

DECLARE @Number NCHAR(7) = 'ABC1234';
DECLARE @LastName NVARCHAR(50) = 'Tanuan';
DECLARE @FirstName NVARCHAR(50) = 'Meyer';
DECLARE @SchoolCode NCHAR(3) = 'BUS';
DECLARE @CampusCode NCHAR(1) = 'W';
DECLARE @Location NCHAR(30) = '3G24';
```

Step 2: Insert all records (explicit transaction)

```
BEGIN TRAN;
    -- add row to table 1
    INSERT INTO Person2
        (number, lastName, firstName)
    VALUES
        (@Number, @LastName, @FirstName);
    -- add row to table 2
    INSERT INTO Employee2
        (number, schoolCode, campusCode, location)
    VALUES
        (@Number, @SchoolCode, @CampusCode, @Location);
COMMIT TRAN;
```

Step 3: Update a record (implicit with rollback)

```
SET IMPLICIT_TRANSACTIONS ON;
-- to see how ROLLBACK works with implicit transactions

DECLARE @Number2 NCHAR(7) = 'ABC1234';
DECLARE @ReportsTo2 NCHAR(7) = 'ABC1234';

UPDATE Employee2
SET reportsTo = @ReportsTo2
WHERE number = @Number2;

-- Rollback will succeed if IMPLICIT_TRANSACTION is ON
ROLLBACK;
```

Concurrency control and transaction isolation levels

Introduction to Isolation Levels – Video Clip

- https://www.linkedin.com/learning/sql-server-performance-for-developers/transaction-isolation

Concurrency control

- Recall that database systems must provide ACID semantics:
  - Atomicity
  - Consistency
  - Isolation
  - Durability
- Recall that all update (INSERT, UPDATE, DELETE, MERGE) statements are themselves atomic
  - That is, either the entire statement modifies, inserts, or deletes all of the intended rows, or the effects of the entire statement are undone and an error is reported

Isolation: the real problem

- Atomicity, Consistency, and Durability are relatively straightforward to implement
  - Atomicity: server keeps an “undo” log of operations to undo
  - Durability: write-ahead logging ensures that the effects of transactions are written to stable persistent storage before control is returned to the user
  - Consistency: server ensures that referential integrity constraints are maintained with each transaction
- But the real problem is **Isolation**:
  - How does the server mimic a situation where, rather than 100’s or 1000’s of connections are concurrently modifying the database, you can pretend there is only one? And yield correct outcomes?

The need for concurrency control

- Concurrently-executing transactions may interfere with each other, producing a result that, overall, is **incorrect** – even if each transaction is correct when executed in **isolation**
  - Example: **Lost updates problem**

  Reference: Paul Larson, CS448 Lecture Notes, University of Waterloo, Winter 1993
- Sample scenario
  - Cam Jordan, a part-time employee currently earns 13K per year. The HR department is giving an increase of 1K (e.g., special pay increase) to Cam's annual salary. Concurrently, Macy (Cam's supervisor) is giving Cam a double in salary to start as a full-time employee. After all the updates are done, should Cam's final salary be (A) 14K (pay increase only)? (B) 26K (double only)? Or is it (C) 27K [26K + 1K] or (D) 28K [14K x 2]?

The four types of concurrency problems

(1) Dirty reads (uncommitted dependencies): Occur when a transaction selects data that isn't committed by another transaction.

(2) Lost updates: Occur when two transactions select the same row and then update the row based on the values originally selected.

(3) Nonrepeatable reads (inconsistent analysis): Occur when two SELECT statements of the same data result in different values because another transaction has updated the data in the time between the two statements.

(4) Phantom reads: Occur when you perform an update or delete on a set of rows when another transaction is performing an insert or delete that affects one or more rows in that same set of rows.

(1) Dirty reads

- One transaction reads uncommitted changes made by another transaction
- Result: (r.field * 2) + 1

Time	Transaction A	Transaction B
0	BEGIN TRANSACTION	BEGIN TRANSACTION
1		Read row r of T
2		r.Field = r.field * 2
3		Write row r of T
4	Read row r of T	
5	r.field = r.field + 1	
6	Write row r of T	
7		COMMIT
8	COMMIT	

(2) Lost updates A: [r.Field + 1]

Time	Transaction A	Transaction B
0	BEGIN TRANSACTION	BEGIN TRANSACTION
1		Read row r of T
2	Read row r of T	
3		r.Field = r.field * 2
4	r.field = r.field + 1	
5		Write row r of T
6	Write row r of T	
7		COMMIT
8	COMMIT	

Given: r.Field = 13K

- [r.Field + 1]

= [13K + 1K]

= [**14K**]

(with pay increase only)

(2) Lost updates B: [r.Field x 2]

Time	Transaction A	Transaction B
0	BEGIN TRANSACTION	BEGIN TRANSACTION
1	Read row r of T	
2		Read row r of T
3	r.field = r.field + 1	
4		r.Field = r.field * 2
5	Write row r of T	
6		Write row r of T
7	COMMIT	
8		COMMIT

Given: r.Field = 13K

- [r.Field x 2]

= [13K x 2]

= [**26K**]

(with double pay only)

(2) Lost updates C: [(r.Field x 2) + 1]

Time	Transaction A	Transaction B
0		BEGIN TRANSACTION
1		Read row r of T
2		r.Field = r.field * 2
3		Write row r of T
4		COMMIT
5	BEGIN TRANSACTION	
6	Read row r of T	
7	r.field = r.field + 1	
8	Write row r of T	
9	COMMIT	

Given: r.Field = 13K

- [(r.Field x 2) + 1]

= [(13K x 2) + 1]

= [**27K**]

(27K with double pay and then pay increase)

CONESTOGA
Connect Life and Learning

(2) Lost updates D: [(r.Field + 1) x 2]

Time	Transaction A	Transaction B
0	BEGIN TRANSACTION	
1	Read row r of T	
2	r.field = r.field + 1	
3	Write row r of T	
4	COMMIT	
5		BEGIN TRANSACTION
6		Read row r of T
7		r.Field = r.field * 2
8		Write row r of T
9		COMMIT

Given: r.Field = 13K

- [(r.Field + 1) x 2]

= [(13K + 1) x 2]

= [**28K**]

(28K with pay increase and then double pay)

CONESTOGA
Connect Life and Learning

(3) Nonrepeatable reads

- One transaction reads a row, another transaction commits a change, and the first transaction reads the row again; the data is different

Note: Application may warn the user and user repeats input (…other work…)

Time	Transaction A	Transaction B
0	BEGIN TRANSACTION	
1	Read row r of T	
2		BEGIN TRANSACTION
3		Update row r in T
4		COMMIT
5	Read row r of T	
6	…other work…	
7	COMMIT	

(4) Phantom reads

- One transaction retrieves a set of rows; another transaction inserts or deletes rows; the first transaction retrieves a set of rows using the same criteria; the result set is different the second time

Time	Transaction A	Transaction B
0	BEGIN TRANSACTION	
1	Subquery 1 on T	
2		BEGIN TRANSACTION
3		Insert row r into T
4		COMMIT
5	Subquery 2 on T	
6	...other work…	
7	COMMIT	

Concurrency Control Schemes

- There are two commonly used concurrency control schemes:
  - Locking
  - Optimistic concurrency control or snapshot isolation
- Microsoft SQL Server uses both locking and snapshot isolation techniques to control transaction isolation behaviour
- Which scheme is used by an application program can be controlled by the application
  - For most schemes, locks are used to prevent anomalies
  - For snapshot isolation, the server makes additional copies of the data as required
  - Tradeoff: **correctness** *versus* **performance**

Concurrency problems prevented by each isolation level

- Syntax:

  SET TRANSACTION ISOLATION LEVEL <isolation level>

- SQL Server's isolation level default of **READ COMMITTED**

Isolation level	Dirty reads	Lost updates	Nonrepeatable reads	Phantom reads
READ UNCOMMITTED	Allows	Allows	Allows	Allows
READ COMMITTED	Prevents	Allows	Allows	Allows
REPEATABLE READ	Prevents	Prevents	Prevents	Allows
SNAPSHOT	Prevents	Prevents	Prevents	Prevents
SERIALIZABLE	Prevents	Prevents	Prevents	Prevents

(a) READ UNCOMMITTED

- READ UNCOMMITTED specifies that statements can read rows that have been modified by other transactions but not yet committed.
- Transactions running at the READ UNCOMMITTED level do not issue shared locks to prevent other transactions from modifying data read by the current transaction.
- READ UNCOMMITTED transactions are also not blocked by exclusive locks that would prevent the current transaction from reading rows that have been modified but not committed by other transactions.
- When this option is set, it is possible to read uncommitted modifications, which are called dirty reads.
- Values in the data can be changed and rows can appear or disappear in the data set before the end of the transaction.

(b) READ COMMITTED

- READ COMMITTED specifies that statements cannot read data that has been modified but not committed by other transactions. This prevents dirty reads.
- Data can be changed by other transactions between individual statements within the current transaction, resulting in non-repeatable reads or phantom data.
- This option is the SQL Server **default**.
- Behaviour is slightly different if snapshot isolation is in effect for other transactions

(c) REPEATABLE READ

- REPEATABLE READ specifies that:
  - statements cannot read data that has been modified but not yet committed by other transactions, and
  - that no other transactions can modify data that has been read by the current transaction until the current transaction completes.
- Shared locks are placed on all data read by each statement in the transaction and are held until the transaction completes.
- Shared locks prevent other transactions from modifying any rows that have been read by the current transaction.
- Other transactions can insert new rows that match the search conditions of statements issued by the current transaction.

(d) SNAPSHOT

- SNAPSHOT specifies that data read by any statement in a transaction will be the version of the data that existed at the start of the transaction.
- The transaction can only recognize data modifications that were committed before the start of the transaction.
- Data modifications made by other transactions after the start of the current transaction are not visible to statements executing in the current transaction.
- The effect is as if the statements in a transaction get a snapshot of the committed data as it existed at the start of the transaction.
- Except when a database is being recovered, SNAPSHOT transactions do not request locks when reading data.
- SNAPSHOT transactions reading data do not block other transactions from writing data.
- Transactions writing data do not block SNAPSHOT transactions from reading data.

(e) SERIALIZABLE

- SERIALIZABLE specifies the following:
  - Statements cannot read data that has been modified but not yet committed by other transactions.
  - No other transactions can modify data that has been read by the current transaction until the current transaction completes.
- Other transactions cannot insert new rows with key values that would fall in the range of keys read by any statements in the current transaction until the current transaction completes.
- Range locks are placed in the range of key values that match the search conditions of each statement executed in a transaction.
- This blocks other transactions from updating or inserting any rows that would qualify for any of the statements executed by the current transaction.
- This means that if any of the statements in a transaction are executed a second time, they will read the same set of rows.