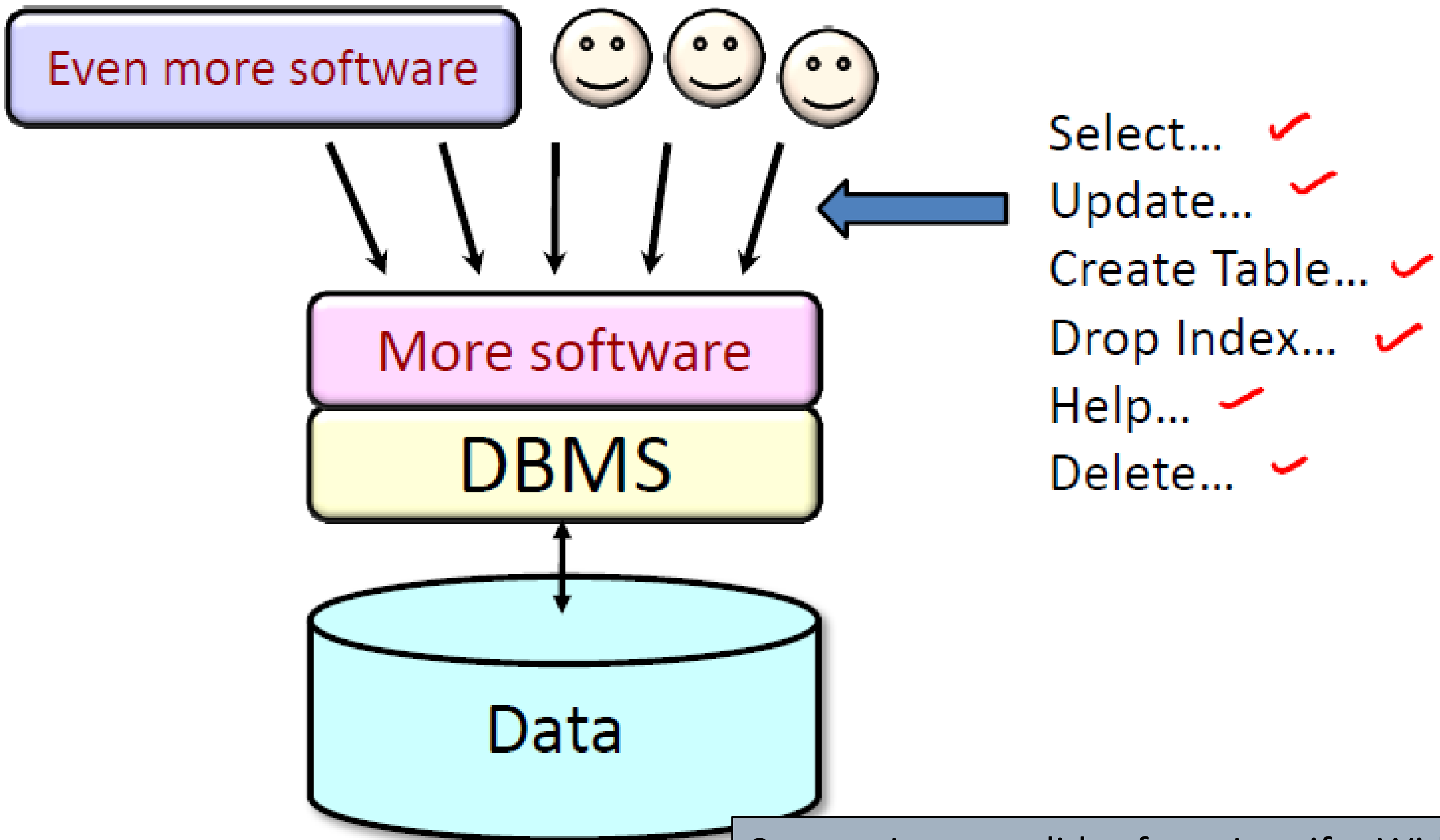
Transaction Concepts through SQL

PM Jat
pm_jat@daiict.ac.in

Why do we study Transactions

- Study of Transactions addresses following two requirements –
  - <u>Enable concurrent access of databases</u> - multiple users need to concurrently access and update databases.
  - <u>Resilience to system failures</u> - While databases are updated system might crash in between.

Concurrent Database Access



Source: Lecture slides from Jennifer Widom

Execution of SQL statements

- SQL statements higher level expressions, and their execution involves execution of

```
UPDATE employee
SET salary = salary + 3000 WHERE ssn = '1234';
```

- Let us say that transactions are executed in a sequqnce of logical operations - Read X, Modify X, Write X.

Execution of SQL statements

```
UPDATE employee
SET salary = salary + 3000 WHERE ssn = '1234';
```

- Using the said model, let us say that above SQL statements is executed as –
  - Read Salary (let us assume it is 50,000),
  - Modify it to 53000 (by adding 3000)
  - Write 53000 to database

- At the end of transaction, database has 53000 as value of “the salary”

Relations used here

- Employee(SSN, Name, Salary, DNO)
- JobApplications(AppNo, ExpectedSalary, HireScore, Experience, SalaryOffered, Hired)

- Account(AccNo, Balance)
- Transaction(AcNo,TS,Description, CrDr, Amuont)

Concurrent execution of (SQL) statements

- Consider execution of following SQL statements from two concurrent clients (users).

C1: **`UPDATE account SET balance = balance + 3000 WHERE accno = 1234;`**

C2: **`UPDATE account SET balance = balance - 5000 WHERE accno = 1234;`**

- DBMS might execute them in interleaved fashion in following order –
  - C1 reads balance (10000)
  - C2 reads balance (10000)
  - C1 modifies it to (13000),
  - C2 modifies to 5000, C2 writes to database 5000,
  - C1 writes to database 13000

Issues with concurrent execution : Case-1

C1: **UPDATE account SET balance = balance + 3000 WHERE accno = 1234;**

C2: **UPDATE account SET balance = balance - 5000 WHERE accno = 1234;**

- DBMS executes them in following order –
  - C1 reads balance (10000)
  - C2 reads balance (10000)
  - C1 modifies it to (13000),
  - C2 modifies to 5000, C2 writes to database 5000,
  - C1 writes to database 13000
- Assuming that before both statements begin their execution, database has balance of 10000 for account no 1234.
- What will be and what should be final value of balance?
  Do you see the problem?

Issues with concurrent execution : Case-2

C1: **`UPDATE employee SET salary=60000 WHERE ssn = 1234;`**

C2: **`UPDATE employee SET dno=4 WHERE ssn = 1234;`**

- DBMS may not be reading and writing only a single value from data on disk. Consider if smallest size DBMS reads and write is a tuple.
- Consider if above two statements are executed in following order –
  - C1 reads the tuple (ssn=1234)
  - C1 modifies the tuple, sets salary to 60000
  - C2 reads the tuple (ssn=1234)
  - C2 modifies the tuple, sets dno=4
  - C1 writes the tuple to the database
  - C2 writes the tuple to the database
- Note the problem?

v1.1

Issues with concurrent execution : Case-3

C1: **UPDATE JobApps SET OFFER = 40000 WHERE AppNo IN (SELECT AppNo FROM JobApps WHERE score > 65);**

C2: **UPDATE JobApps SET score = 1.1*score WHERE exp >= 5;**

- C1:
```
for each job_app x from JobApps
    read tuple x;
    If x.score > 65 then
        x.offer = 40000;
        write x;
```
- C2:
```
for each job_app x from JobApps
    read tuple x;
    x.score += 1.1*x.score;
    write x;
```

v1.1

Issues with concurrent execution : Case-3

C1: `UPDATE salary SET salary = 1.1*salary;`

C2:

```
SELECT avg(salary) FROM employee WHERE ;
SELECT avg(salary) FROM employee;
```

- C1:
```
for each employee e from EMP
        read tuple e;
        e.salary = 1.1 * e.salary;
        write e;
```
- C2:
```
s=0;n=0;
for each employee e from EMP
        read tuple e;
        s += e.salary;
        write e;
```

Issues with concurrent execution : Case-4

C1: **INSERT INTO ex_emps SELECT .. FROM emp WHERE ex_flag;**

DELETE FROM emp WHERE ex_flag;

C2: **SELECT count(*) FROM emps;**

SELECT count(*) FROM ex_emps;

- Note the problem?

Issue - System Failures

- Suppose, we need to transfer amount of 5000 from account number 1011 to 2312; and we have following database updates to log the transaction and update the balances accordingly.

```
INSERT INTO transaction VALUES (1011, now,
   'Transferred to A/c 2312','Debit',5000);
INSERT INTO transaction VALUES (2312, now,
   'Transferred from A/c 1011','Credit', 5000);
UPDATE account SET balance=balance–5000 WHERE
   acno = 1011;
UPDATE account SET balance=balance+5000 WHERE
   acno = 2312;
```

- What if system crashes after executing 3rd statement ? We ought to execute either all statements or none ?

Conclusion drawn from issues

- Uncontrolled sequencing of operations for execution may be bring database in inconsistent state!!
- Partial execution of a set of statements may bring database in inconsistent state.

Solution

- Control the concurrency
  - Order the Read/Write operations from multiple clients such that they appear to be running in isolation

- Guarantee executing sequence of related operations (Transactions) from a client in “all or nothing” manner, regardless of failures

- “Transaction” are to address both the above concerns

Notion of “Transaction”

- A transaction is a sequence of one or more SQL operations treated as a unit

- A sequence of SQL operations, performing a single application task; for example
  - Transfer amount from one account to another account; a sequence of related sql statements form a transaction
  - Save an Invoice; a sequences of statements that typically makes entry into multiple invoice tables; and update stock tables form a Transaction

SQL commands related to Transactions

- BEGIN TRANSACTION
- COMMIT
- ROLLBACK
- SET TRANSACTION
  - ISOLATION LEVEL
  - ACCESS MODE

ACID properties of Transaction

- ACID is acronym for
  - **A**tomicity
  - **C**onsistency
  - **I**nsolation
  - **D**urability

- These are desirable characteristics of Transaction Processing

ACID - **Isolation**

- If we execute transaction in isolation, that is no interleaving, DBMS let finish one transaction before taking up turn of another.
- That means no interleaving; that we can not afford – wait time, processor under-load, infinite loops in a transaction etc., etc.
- This conflicting situation brings in notion of **Serializability**.

ACID – **Isolation (Serializability)**

- Serializability: Operations may be interleaved, but execution must be equivalent to *some* **serial** (sequential) **order** of all transactions
- That is suppose there are two transactions T1 and T2 are executing concurrently; interleaving of operations from T1 and T2 should be such that the result is equivalent to either T1 followed by T2 executes in isolation, or T2 followed by T1.
- Notationally we say that "**T1;T2**" or "**T2;T1**".
- So basically Isolation property requires transaction execution to ensure serializability.

ACID – **Isolation (Serializability)**

- So basically Isolation property requires transaction execution to ensure *Serializability*.

- With this promise, we can see that issues related to concurrency discussed earlier all gone.

- Case 1:

C1: **`UPDATE account SET balance = balance + 3000 WHERE accno = 1234;`**

C2: **`UPDATE account SET balance = balance - 5000 WHERE accno = 1234;`**

- No issue when executed as either C1;C2 or C2;C1

ACID – **Isolation (Serializability)**

- In case-2, and case-3 also, serializability ensures correct result.

Case-2

C1: **`UPDATE employee SET salary=60000 WHERE ssn = 1234;`**

C2: **`UPDATE employee SET dno=4 WHERE ssn = 1234;`**

ACID – **Isolation (Serializability)**

- However in caser-3, different order produces different result, and might be desirable to other.
- Serializability does not guarantee the order of execution, it has to be controlled by the programmer

Case-3

C1: **`UPDATE JobApps SET OFFER = 40000`**

```
WHERE AppNo IN (SELECT AppNo FROM JobApps WHERE
score > 65);
```

C2: **`UPDATE JobApps SET score = 1.1*score`**

```
WHERE exp >= 5;
```

ACID – **Isolation (Serializability)**

- Case 4 also produces correct result; and this case also order will matter the result.

C1:
```
INSERT INTO ex_emps SELECT .. FROM emp WHERE ex_flag;
DELETE FROM emp WHERE ex_flag;
```
C2:
```
SELECT count(*) FROM emps;
SELECT count(*) FROM ex_emps;
```

ACI**D** - Durability

- This characteristics requires that If system crashes after transaction commits; all effects of transaction reflects in database.

- In real systems, DBMS might acknowledge the client for a commit request, and a system failure occurs before effects of committed transaction are updated on disk.

- A transaction processing system should prevent this and ensure --> Durability.

ACID - Atomicity

- A transaction is executed in “all or nothing” manner.
- A transaction is never half executed.
- DBMS ensures this by maintaining appropriate logs, and by recovery process.

ACID - Consistency

- Each transaction from each client can assume that all constraints hold when transaction begins.
- Transaction execution should ensure that all constraints hold true after transaction commits.
- Serializability and atomicity takes care of this property.

Repeat

- Study of Transactions addresses following two requirements –
  - Enable concurrent access of databases - multiple users need to concurrently access and update databases.
  - Resilience to system failures - While databases are updated system might crash in between.

- SQL Transaction Isolation Levels

SQL Isolation Levels

- While serializability is desirable; it has following side effects -
  - Brings in additional overhead (processes that ensure serializable execution schedules)
  - Provides reduced concurrency; Serialization algorithms may ask some transaction to wait and aborts

- However Serializability may not required all the time, particularly for reads; therefore SQL allows specifying a transactions to execute “weaker isolations”.

SQL Transaction Isolation Levels

- SQL standard specifies four Isolation levels in which a transactions can be executed
  - Serializable
  - Repeatable Read
  - Read Committed
  - Read UnCommitted

Weaker "Isolation Levels"

weak

Read Uncommitted

Read Committed

Repeatable Read

strong

Serializable

↓ Overhead ↑ Concurrency

↓ **Consistency Guarantees**

Source: Lecture slides from Jennifer Widom

SQL Transaction Isolation Levels

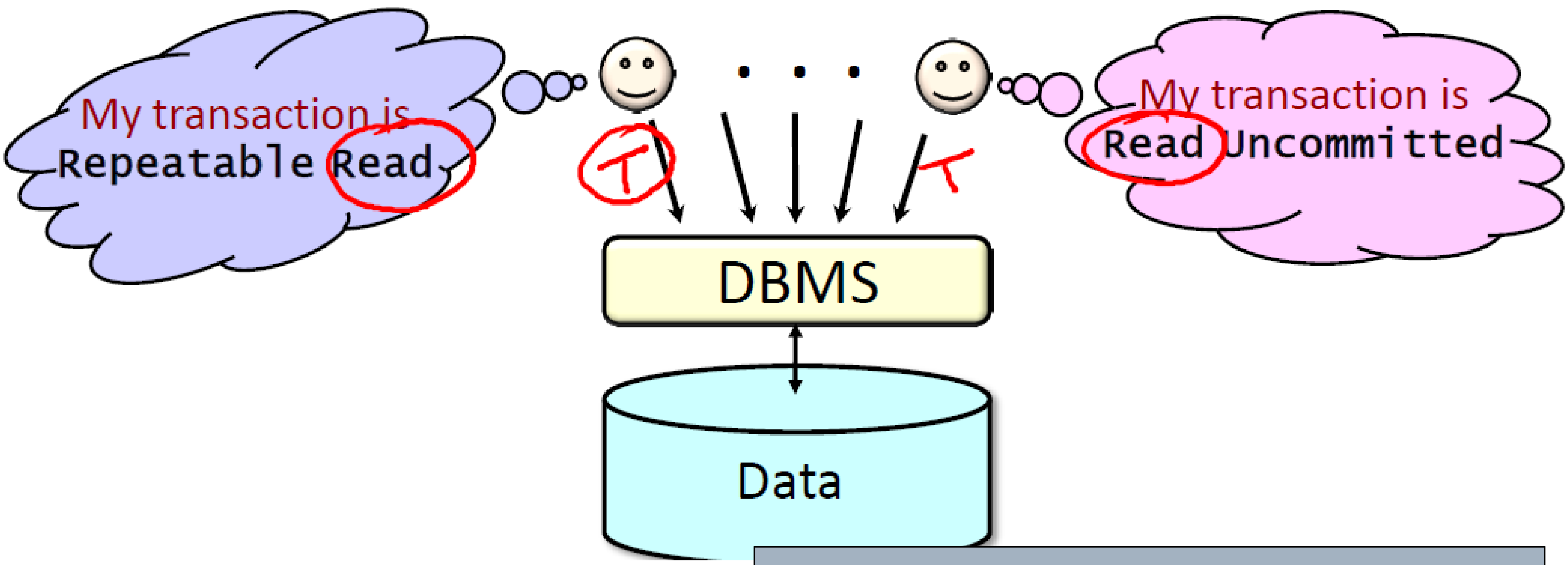
- Specified for the transaction
- Different transaction may execute at different isolation levels
- SQL has SET Transaction command to specify isolation level, and done as following -

```
Set Transaction Isolation Level <isolation level>;
```

SQL Transaction Isolation Levels

Isolation Levels

- Per transaction
- "In the eye of the beholder"



Source: Lecture slides from Jennifer Widom

Transaction isolation levels in SQL-92

- **`Serializable`**
- **`Repeatable read`** —
  - only committed records to be read,
  - repeated reads of same record must return same value; however, a transaction may not be serializable- it may find some records inserted by other concurrent transactions.
- **`Read committed`** –
  - only committed records can be read,
  - but successive reads of same record may return different (but committed only) values.

Transaction isolation levels in SQL-92

- **`Read uncommitted`** –
  - Even uncommitted records may be read.
  - Dirty read may occur

- As per SQL standard, **Serializable** is the default isolation level.

- However a RDBMS may have different level as default. For example PostgreSQL has **read committed** as default isolation level.

- Problems with lower isolation levels

Isolation Level – **Read Uncommitted**

T1:

```
UPDATE emp SET salary = 1.1*salary;
ROLLBACK; //COMMIT
```

T2: `SELECT average(salary) FROM emp;`

- Suppose T2 runs at “Read UnCommiited” level, then it allows reading data from T1 even before it commits. If T1 decides to abort then report of T2 is incorrect.
- Even when T1 commits, T2 might read some old data before T1 updates and some updated that also undesirable.
- This phenomena of reading uncommitted data is called as “Dirty Read”.

Isolation Level – **`Read Uncommitted`**

- A transaction running at **`Read Uncommitted`** isolation level <u>may perform dirty reads</u>

- Note that in previous example if T2 performs dirty then it is not ensuring serializability; that is

  Neither "T1;T2" nor "T2;T1"

Isolation Level – `Read Committed`

- A transaction running at **Read Committed** level does not read updates from uncommitted transaction.
- Consider concurrent execution of following transaction

T1: `UPDATE emp SET salary = 1.1*salary;`

`COMMIT;`

T2: `SELECT average(salary) FROM emp;`

`SELECT max(salary) FROM emp;`

- If first statement of T2 executes (in Read Committed level) while T1 executes its UPDATE; it reads from old data.
- And by the time T2 starts executing second statement, T1 commits, so second statements reads data updated by T1

Isolation Level – **Read Committed**

T1: `UPDATE emp SET salary = 1.1*salary;`
`COMMIT;`
T2: `SELECT average(salary) FROM emp;`
`SELECT max(salary) FROM emp;`

- The said execution is neither T2;T1 nor T1;T2.
- Execution of transaction at this level does not perform dirty read, still does not guarantee serializability.
- It is due to insufficient isolation level

Isolation Level – **Repeatable Read**

- A transaction reads same value if a data item is read multiple times; that is repeated read of a item gets same value in a transaction.
- In previous example, problem was due to non repeatable reads. Now if T2 in following concurrent execution runs at "Repeatable Read; read of salary in both statements of T2 will be same ==> results serializable execution

T1: **UPDATE emp SET salary = 1.1*salary;**
COMMIT;
T2: **SELECT average(salary) FROM emp;**
SELECT max(salary) FROM emp;

Isolation Level – **`Repeatable Read`**

- Consider concurrent execution of following transactions-

T1: **`INSERT INTO emp ..;`** //say inserts 10 tuples

`COMMIT;`

T2: **`SELECT average(salary) FROM emp;`**

`SELECT average(salary) FROM emp;`

- If first statement of T2 executes (in Repeatable Read) while T1 executes its UPDATE; it reads from old data.
- And by the time T2 starts executing second statement, T1 commits, so second statements reads data updated by T1
- This execution again will not achieve serializability?
- Repeatable reads sees rows inserted by other concurrent transactions, called “phantom rows”

Isolation Level – **Repeatable Read**

- Consider concurrent execution of following transactions-

T1: **UPDATE emp SET salary = 1.1*salary;**

UPDATE emp SET dno=4 WHERE ssn=1234

COMMIT;

T2: **SELECT average(salary) FROM emp;**

SELECT count(*) FROM emp WHERE dno=4;

- If first statement of T2 executes (in Repeatable Read) while T1 executes its UPDATE; it reads from old data.
- And by the time T2 starts executing second statement, T1 commits, so second statements reads data updated by T1
- This execution may not achieve serializability? If concurrent transaction

Isolation Level - Summary

	dirty reads	nonrepeatable reads	phantoms
Read Uncommitted	Y	Y	Y
Read Committed	N	Y	Y
Repeatable Read	N	N	Y
Serializable	N	N	N

weak

strong

Source: Lecture slides from Jennifer Widom

Isolation Level - Summary

- Standard default: Serializable
- Weaker isolation levels
  - Increased concurrency + decreased overhead = increased performance
  - Weaker consistency guarantees
  - Some systems have default Repeatable Read
- Isolation level per transaction and “eye of the beholder”
  - Each transaction’s reads must conform to its isolation level

Source: Lecture slides from Jennifer Widom

- Source: Lecture slides from Jennifer Widom