12. Transaction Processing

Most of content here is borrowed from following sources-
>> Lecture slides of Jennifer Widom, Stanford
>> Book Elmasri/Navathe

Transaction – Motivation

Following are two reasons of studying transactions.

1. Enable concurrent access of databases: DBMS receives data manipulate requests from multiple concurrent users. If we allow executing these requests in arbitrary order (or even in the order they arrive), it may either lead to wrong reporting, or more seriously may lead database becoming inconsistent.
2. Safeguard database against system failures - while databases are getting updated system might crash in between leaving database in inconsistent state. This problem is basically due to partial execution of a task on database.

We needs to have certain techniques implemented in DBMS to ensure that above two concerns are addressed. Here we study such techniques.

Transaction

What a "Transaction" is?

Consider relation model of databases; database operations are specifying using SQL. SQL is language to define operations on relations. However relational operations do not represent "business operations". Following are some examples of business tasks (operations)-

- Save an Order
- Transfer Fund from one account to another account
- Withdraw money
- Receive Money
- Give salary Raise
- Add a Customer

A business operation may require more than one SQL statements to be submitted to the DBMS. This is what brings in notion of Transaction. A transaction can be defined as a "sequence of one or more SQL statements" representing a business task.

While executing a business operation, that is a sequence of SQL statements (Transactions), we have two conflicting challenges-

1. We need to execute "operations" from multiple users concurrently, i.e. execution of concurrent requests from multiple users should go simultaneously; may be in "interleaved" fashion.
2. We do not want partial execution of any "Transaction"

Data Model for explaining transaction processing issues and techniques

We have "relations" at logical level, and have SQL to manipulate relations, and hence databases.

However "relation" data are stored on disks as "records". We perform lower level input/output operations for executing SQL statements. There are techniques to locate desired data records on disk blocks. For answering of queries, data from relevant blocks are brought into primary memory; may further process the data and present to the user.

For update operations, block(s) containing target data records are brought in the primary memory; necessary updates are done at appropriate places in blocks, and finally blocks are saved on disks at appropriate place aligned with corresponding file organization and access paths.

For transaction processing purposes, we use different data model, defined as following-

- A database is a collection of "named data items".
- Granularity of data can be a field, a record, a disk block, or even a whole file.
- Operations are just two "read" and "write"; and assume that all SQL statements are executed using these read/write primitives.

Let us also present an understanding of said "read" and "write" operations, as following-

mx = READ(data_x): reads database item **data_x,** and say it does following -

- Find block of **data_x** on disk
- Load block into buffer, if not already there
- Returns value of **data_x**

WRITE(data_x, mx): write memory value **mx** to **data_x** on database, and say it does following-

- Find address of **data_x** on disk
- Load block into buffer, if not already there
- Update **data_x** in buffer
- Write appropriate block into disk: may not be done immediately

For example consider following SQL statement-

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

In terms of said operations, let us say this SQL statement is executed as following

```
x = READ(SALARY1234);
x := x + 3000;
WRITE(SALARY1234 , x);
```

Let us take another example of a sequence of SQL statements expressed in terms of READ/WRITE statements. Let us say below is a transaction that transfers 3000 from acno 1011 to 5756-

```
UPDATE account SET balance = balance-3000 WHERE acno = 1011;
UPDATE account SET balance = balance+3000 WHERE acno = 5756;
```

The transaction expressed in terms of read/write operations:

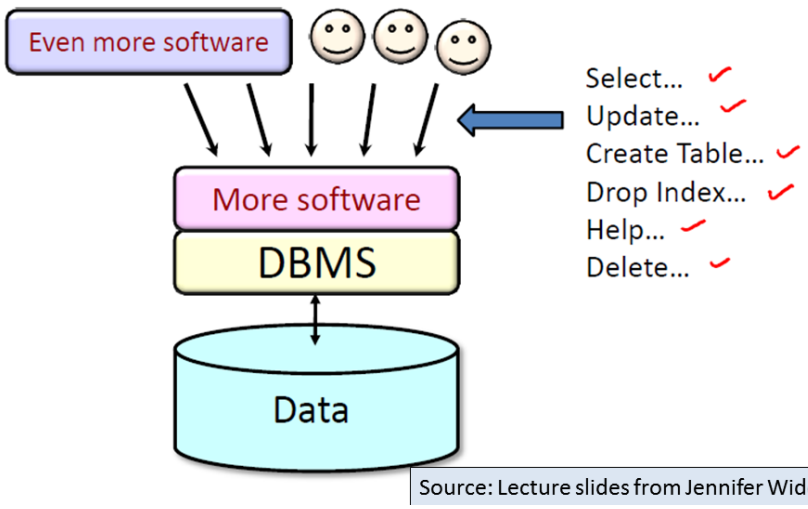
```
b = READ(BALANCE1011);
b := b - 3000;
WRITE(BALANCE1011, b);
b = READ(BALANCE5756);
b := b + 3000;
WRITE(BALANCE5756, b);
```

More operations on a transaction

1. BEGIN TRANSACTION (Indicates that transaction begins)
2. COMMITT TRANSACTION (Make the transaction updates permanent on disk files)
3. ROLLBACK TRANSACTION (Undo the updates of a transaction)

Problems due to concurrent Access

Any real database is accessed by multiple users; and in a typical OLTP environment, some users are reading data (answering select queries), and some clients are updating database. Diagram below should capture the scenario.



Source: Lecture slides from Jennifer Widom

3. Let DBMS include operations from multiple transactions and execute in interleaved serial order. Below is an example depicting interleaved "execution schedule" of DBMS-

Suppose following three transactions are executing concurrently- T1(s1, s2, s3, s4), T2(s8, s9, s10), T3(s11, s12, s13, s14); and typically operation may be interleaved as following-

…, s1,s2,s8,s11,s9,s12,s3,s4,s13,s10,s14,…

Note that while interleaving statements from multiple transactions, order of statements from a transaction does not change, however instructions from other transaction get inserted in between.

Let us see some contexts to demonstrate problems arising due to concurrent access of databases. Take a note of relations given below, that will be used in following discussions-

- Employee(SSN, Name, Salary, DNO)
- JobApplications(AppNo, ExpectedSalary, HireScore, Experience, SalaryOffered, Hired)
- Account(AccNo, Balance)
- Transaction(AcNo, TS, Description, CrDr, Amuont)

Case-1

Say following two SQL statements are simultaneously submitted by two clients C1 and C2; both are updating balance of account no 1234. Let us say current balance is 10000.

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

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

Let us say, DBMS executes them in interleaved fashion in order as given below –

[C1]:	x=READ($BALANCE_{1234}$)	[x=10000]
[C2]:	y=READ($BALANCE_{1234}$)	[y=10000]
[C1]:	x := x + 3000	[x=13000]
[C2]:	y := y - 5000	[y=5000]
[C2]:	WRITE ($BALANCE_{1234}$,y)	[balance becomes 5000 in DB]
[C1]:	WRITE ($BALANCE_{1234}$,y)	[balance becomes 13000 in DB]

Correct update should be: [balance=8000]

Note the problem? Update of one transaction lost ==> LOST UPDATE problem

Case-2

Say following two statements are simultaneously submitted by two clients C1 and C2;

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

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

Attribute level read/write is too low granularity in database operations, therefore let us say it done at record or tuple level, and let us say execution schedule generated is as following –

[C1]: x = READ(EMP_{1234}) [x=<1234, Anil, 3, 50000>]
[C1]: x.salary := 60000 [x=<1234, Anil, 3, 60000>]
[C2]: y=READ(EMP_{1234}) [y=<1234, Anil, 3, 50000>]
[C2]: y.DNO=4 [y=<1234, Anil, 4, 50000>]
[C1]: WRITE (EMP_{1234},x) [<1234, Anil, 3, 60000>] //tuple in DB
[C2]: WRITE (EMP_{1234},y) [<1234, Anil, 4, 50000>] //tuple in DB
Correct update should be: [<1234, Anil, 4, 60000>]
Note the problem? What should actual state of the tuples in correct processing?
Update of one transaction lost ==> LOST UPDATE problem

Case-3

Say following two statements are simultaneously submitted by two clients C1 and C2;

```
C1:  UPDATE employee SET salary=60000 WHERE ssn = 1234;
     UPDATE employee SET DNO=3 WHERE ssn = 2345;
     ROLLBACK;

C2:  UPDATE employee SET salary=salary+5000 WHERE ssn = 1234;
```

In this case let us say DBMS is reading whole record (tuple), reading an attribute, let us say too a low granularity in database operations –

[C1]: x = READ(EMP_{1234}) [x=<1234, Anil, 3, 50000>]
[C1]: x.salary := 60000 [x=<1234, Anil, 3, 60000>]
[C1]: WRITE (EMP_{1234},x) [<1234, Anil, 3, 60000>] //tuple in DB
[C2]: y=READ(EMP_{1234}) [y=<1234, Anil, 3, 60000>]
[C2]: y.salary := y.salary+5000 [y=<1234, Anil, 3, 65000>]
[C1]: z = READ(EMP_{2345}) [z=<2345, Mukesh, 2, 40000>]
[C1]: z.DNO := 3 [z=<2345, Mukesh, 3, 40000>]
[C1]: ROLLBACK [<1234, Anil, 3, 50000>], [<2345, Mukesh, 2, 40000>]/in DB
[C2]: WRITE (EMP_{1234},y) [[<1234, Anil,3, 65000>] //tuple in DB]
[C2]: COMMIT
Correct update should be: [<1234, Anil, 3, 55000>]

Note the problem? What should actual state of the tuples in correct processing?
==> DIRTY READ

Case-4

Say following two statements are simultaneously submitted by two clients C1 and C2;

C1: UPDATE salary SET salary = 1.1*salary;

C2: SELECT avg(salary) FROM employee;

Say above SQL statements are translated into read/write and executed in interleaved manner as following [this translation is indicative, may not actually done this way]

```
C1:   for each employee e from EMP
          e = READ(EMP_e);   //reads a tuple
          e.salary := 1.1 * e.salary;
          WRITE(EMP_e, e);   //write the tuple
C2:   sum=0; n=0;
      for each employee e from EMP
          e = READ(EMP_e);   //read a tuple
          if ( e.salary <> null )
                sum += e.salary;
                n += 1;
      avg = sum / n;
```

Note the problem? INCONSISTENT READ;
C2 either should have read all old salary or all new salary!

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 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

ACID properties of Transaction

ACID is acronym for

- **A**tomicity
- **C**onsistency
- **I**nsolation
- **D**urability

These are desirable characteristics of Transaction Processing. If ensured, concurrent execution of transaction would not have any undesirable consequences.

ACID – ISOLATION

- If we execute transaction in isolation, that is no interleaving, that is
  - Let DBMS finish one transaction before taking up turn of another.
- That means no interleaving; that we cannot afford
  - wait time, processor under-load, infinite loops in a transaction etc., etc.

This conflicting situation brings in notion of **Serializability**.

What is Serializability?

- Operations may be interleaved, but execution must be equivalent to *some* **serial** (sequential) **order** of all transactions.
- For example, suppose there are two transactions T1 and T2 are executing concurrently; having operations (s1,s2,s3) and (s11, s12) respectively.
- Serial execution is either all of T1 is executed before any operation of T2 or all operations from T2 is executed before any operation from T1. These are called as "Serial Execution Schedule"
- First case can be expressed as <T1; T2>, that is <s1, s2, s3; s11, s12>, and second case as <T2;T1>, that is <s11, s12; s1, s2, s3>.

- Interleaving of operations from T1 and T2 should be such that the result is equivalent to some "serial execution schedule". We call such an execution as "Serializable Schedule".
- Serializable schedules are schedule that are equivalent to some serial schedule.
- So basically Isolation property requires transaction execution to ensure serializability.

ACID – Durability

- This characteristic 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, but committed updates are done only on buffer, and may not be immediately flushed on disk
- If system fails at this point of time content of buffer may get lost, this implies committed updates of committed transaction did not reflect in DBMS, this is undesirable
- Durability characteristic requires, "commits are durable" irrespective of system failures.

ACID - Atomicity

- A transaction is executed in "all or nothing" manner.
- A transaction is never half executed.
- To ensure this, it may be undesirable to write anything on the disk until transaction commits; however this may not be feasible. Buffer management techniques needs to flush changes to the disk when memory is not sufficient.
- Failure at this stage might leave a transaction half done on DBMS.

Durability and Atomicity are ensured by DBMS maintaining "logs of all updates" on database; and Durability and Atomicity is ensured by "redoing" certain operations (typically of committed transaction) and from logs. To ensure atomicity again systems uses logs and undoes typically undoes operations of uncommitted transactions.

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.

Serializability

To Repeat: Serializability is a mechanism for implementing Isolation property of ACID.

Before proceeding, let us have more compact representation of an execution "schedule" that will also be used for understanding further concepts. Consider following concurrent transactions T1 and T2, and note order of their interleaved operations.

T1		**T2**	
	mx1=READ(X)		
	mx1 := mx1 – N		
			mx2=READ(X)
			mx2 := mx2 + M
	WRITE(X,mx1)		
	my1=READ(Y)		
			WRITE(X, mx2)
	my1 := my1 + N		
	WRITE(Y, my1)		

Let us express transactions and schedule as following-

T1: mx1=r1(X); mx1=mx1-N; w1(X, mx1); my1=r1(Y);my1=my1+N; w1(Y,my1);
T2: mx2=r2(X); mx2=mx2+M; w2(X, mx2);

Let the number with r and w, represents transaction identification.

Serial Schedule-1 S1:
mx1=r1(X); mx1=mx1-N; w1(X, mx1); my1=r1(Y); my1=my1+N; w1(Y,my1);
mx2=r2(X); mx2=mx2+M; w2(X,mx2); ==> T1;T2

Serial Schedule-1 S2:
mx2=r2(X); mx2=mx2+M; w2(X, mx2); mx1=r1(X); mx1=mx1-N;
w1(X, mx1); my1=r1(Y); my1=my1+N; w1(Y,my1); ==> T2;T1;

Interleaved schedule-1 S3:
mx1=r1(X); mx1=mx1-N; mx2=r2(X); mx2=mx2+M; w1(X, mx1); my1=r1(Y);
my1=my1+N; w2(X, mx2); ==> equivalent to S1 or S2?

Serializability requires that S3 should be equivalent to one of above S1 or S2?

Checking for Serializability

How do check whether an execution schedule is serializable?

One way is – we see if result produced by executed schedule is equal to computed result for some "serial schedule". For example if we look at final result of S3, and if that is equal to either of S1 and S2, then we say that it is equivalent to that. However this is not practical for two reasons -

1. It may be just by chance
2. It requires computing results before executing the schedule. DBMS, to determine go ahead about a schedule, needs to know all forthcoming operations in advance? This is not feasible.

Second, and often used is, we have some rules (or algorithms) that can detect the non-serializability if "operation from a transaction is under consideration" is included in the schedule? Next we see a formal technique for checking serializability.

Conflict Serializability

A common techniques used for checking a schedule is Serializable or not is "Conflict Serializability".

Intuition behind Conflict Serializability is that orders of all "Conflicting operations" from participating transactions are in same order.

What are conflicting operations?

It can be understood that concurrent access problem would likely to occur only when one of the operation in a transaction is write operations. With this intuition, conflicting operations are defined as following.

Two operations in a schedule are said to be conflicting, if they satisfy all of following conditions -

- Belong to two different transactions
- They access the same data items
- At-least one of them is write operations

Following are conflicting operations (that is if an data item is accessed by two transaction, and one of them is writing then it is a conflicting operation)-

- r1(X) and w2(X)
- w1(X) and w2(X)

Following are non-conflicting operations

- r1(X) and r2(X) //both are read
- w1(X) and w2(Y) //both work on different data item

By using this notion of conflicting operations, we define a serializability as following –

> If order of all conflicting operations in a schedule is same as in some serial schedule, then we can say that schedule in serializable. We call such serializability as "conflict serializability".

Now let us apply this check in schedule S3

S3: mx1=r1(X); mx1=mx1-N; mx2=r2(X); mx2=mx2+M; w1(X, mx1);
my1=r1(Y); my1=my1+N; w2(X, mx2);

Following conflicting operations and there order:

(1) r1(X); w2(X)
(2) r2(X); w1(X)
(3) w1(X); w2(X)

Conflicting operation pair (1) and (3) are in the order of T1; T2 while operation pair (2) in the order of T2;T1. Hence they are not in same order and hence the schedule is not serializable.

Note that only read/write operation matter in serializability test.

Also it does not matter what value is getting read and written; therefore we further simplify schedule expression. Following are representations of S1, S2, and S3:

S1: r1(X); w1(X); r1(Y); w1(Y); r2(X); w2(X);
S2: r2(X); w2(X); r1(X); w1(X); r1(Y); w1(Y);
S3: r1(X); r2(X); w1(X); r1(Y); w2(X);

And again, with shrunk representation of schedule, you can establish that S3 is not serializable!

Now let us consider another schedule-

S4: r1(X); w1(X); r2(X); w2(X); r1(Y); w1(Y);

Following conflicting operations in S4 and there order:

(1) r1(X); w2(X)
(2) w1(X); r2(X)
(3) w1(X); w2(X)

Now all conflicting operations are in the order of T1; T2. Therefore S4 schedule is equivalent to serial schedule S1, i.e. T1; T2

Testing Conflict Serializability

- A directed graph is built for checking if a schedule is conflict serializable.
- Let us call that Precedence Graph or Serialization Graph.
- It consists of n Nodes T1, T2, … Tn, and a set of edges E1, E2, … Em
- Here edges are such that an edge Ei, say is Tj → Tk, and this denotes that an operation appears in Tj before a conflicting operation in Tk.

- If there is any cycle is formed in the graph then, the schedule is not serializable.
- A cycle can be defined as: if starting and ending node for a sequence of edges is same, then graph forms a cycle.

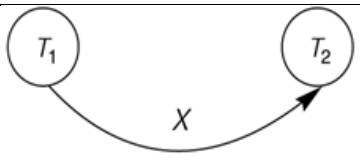
Algorithm for Testing Conflict Serializability

- For each transaction Ti participating in schedule S, create a node labeled Ti in the precedence graph.
- For each case in S where Tj executes a read_item(X) after Ti executes a write_item(X), create edger Ti → Tj
- For each case in S where Tj executes a write_item(X) after Ti executes a read_item(X), create edger Ti → Tj
- For each case in S where Tj executes a write_item(X) after Ti executes a write_item(X), create edger Ti → Tj
- The schedule S is serializable only if, and only if the precedence graph has no cycle.

Precedence graph for our S1 through S4

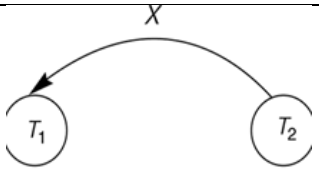
S1: r1(X); w1(X); r1(Y); w1(Y); r2(X); w2(X);

Conflicting operations (T1;T2)
(1) r1(X); w2(X)
(2) w1(X); r2(X)
(3) w1(X); w2(X)



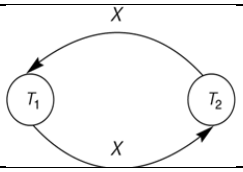
S2: r2(X); w2(X); r1(X); w1(X); r1(Y); w1(Y);

Conflicting operations (T2;T1)
(1) w2(X); r1(X)
(2) r2(X); w1(X);
(3) w2(X); w1(X)



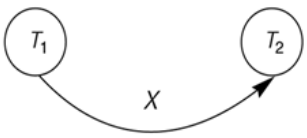
S3: r1(X); r2(X); w1(X); r1(Y); w2(X, mx2);

Conflicting operations
(1) r1(X); w2(X)
(2) r2(X); w1(X)
(3) w1(X); w2(X)



S4: r1(X); w1(X); r2(X); w2(X); r1(Y); w1(Y);

Conflicting operations (T1;T2)
(1) r1(X); w2(X)
(2) w1(X); r2(X)
(3) w1(X); w2(X)



With this understanding of serializability, let us look at our problem examples; see how they can be handled, so that problems do not occur!

Case-1 and Case-2; having LOST UPDATE problem and go as following

T1 read tuple t
T2 reads tuple t
T1 write t after necessary modification
T2 writes t after necessary modification

Create precedence graph? Does it form cycle?

Solution: if we defer T2's read, and allow T1 write before T2 can read the tuple.

Case-4:

- If we ensure conflicting operations in one of other T1; T2 or T2; T1; then there would not be any problem.
- That is - T2 either reads all tuples before T1's update (equivalent to T2;T1); or reads all tuples after T1's update (equivalent to T1;T2)
- That is Query in T2 will either include either all salary old or all new!

Note: Serializability does not guarantee order of equivalence, it just guarantees, serializability.

ISOLATION property of ACID is to ensure execution of SERIALIZABLE schedule.

Why Serializability is important?

- We cannot afford to run serial schedule.
- We need to interleave operations from simultaneous transactions.
- We interleave, but interleave such that they are equivalent to some serial schedule.
- We call such schedule as serializable.

How serializability is ensured?

Serializability is basically ensured by ordering the operations such that DBMS executes schedules in serializable manner.

DBMS module responsible for this is called "Concurrency Control" module. It implements certain concurrency control techniques. Even though it is heavily influenced by appropriate concurrency control techniques, essence of its typical working is as following -

- DBMS ready accepting operation requests (say read and write) from various transactions/clients, and put them in schedule. [Assume that once an operation has been put in a schedule, it is assumed as executed unless it is aborted (rolled back)].
- Before putting an operation in schedule DBMS ensures that resultant schedule will be serializable. There are various techniques for this (Locking, Time Stamp Ordering, etc.)
- If DBMS detects that if an operation under consideration leads to non-serializability, DBMS does one of following (and that depends on what algorithm it is based on, and so) –

- Ask concerned transaction to wait; transaction goes in wait mode. DBMS definitely will notify the waiting transactions when it can accept the request
- Ask requesting transaction to abort, because it may not possible that request to be executed as per serialization requirements.

Recoverability

Recoverability – systems ability to recover from crashes.

Crashes may leave database –

- Committed transaction not physically saved on disks, or
- Updates of uncommitted transactions (or partially done transactions) are physically saved on disks

Both are undesirable; therefore when system restarts, it should ensure-

- Updates of committed transaction reflected on database on disks
- Updates of uncommitted is not reflected on disks

To simplify recovery process there are certain basic conditions ensured while generating schedules.

Recovery Principle

- If we follow a discipline that a transaction T cannot commit until all transactions from which T reads (reads value written by other concurrent transactions), are committed.
- How does this help -
  - Suppose T2 reads from writes of T1, and T1 aborts; T2 also has to abort;
  - More serious if T2 is committed;
  - More serious if T3 reads from writes of T2; if T1 aborts; T2 has to abort, and hence T3 as well; leads to a "**cascaded roll-backs**"
- With assumption of "recovery principle", we limit cascaded aborts only up-to uncommitted transactions.
- This principle also ensures us that we will never have to UNDO a committed transaction.
- "Recovery principle" principle also simplifies recovery process.
- The "recovery system" of DBMS should ensure that the database gets effects of all committed transactions and none of the uncommitted transactions.

Recoverable Schedule:

- A schedule is recoverable, in which no transaction T commits, till all transactions from which T reads are committed or aborted (i.e. transactions that have written data

that T reads, are committed). [Note if any of transaction from which T reads aborts, T should also abort]

Note if transactions never abort; recovery is rather easy. Since all transactions eventually commit - DBMS can simply executes database operations as they arrive.

Primary responsibility of recoverability is to ensure "Durability" and "Atomicity" properties of ACID. DBMS maintains "logs of all updates" on database; and Durability and Atomicity is ensured by "redoing" certain operations (typically of committed transaction) and from logs. To ensure atomicity again systems uses logs and undoes typically undoes operations of uncommitted transactions

We will see recovery process later!

SQL commands related to Transactions

BEGIN TRANSACTION
COMMIT TRANSACTION
ROLLBACK TRANSACTION
SET TRANSACTION

- ISOLATION LEVEL
- ACCESS MODE

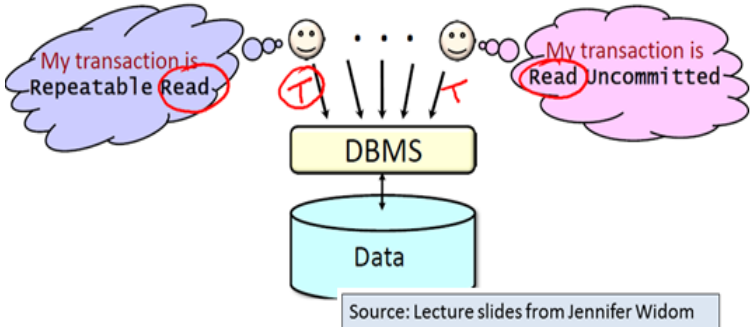
Syntax of specifying Isolation levels -

Set Transaction Isolation Level *<isolation level>*;

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 other transaction to wait, and even aborts in some cases.
- However Serializability may not be required all the time, particularly for reads. Therefore RDBMS implementation also "weaker isolations".
  - Idea of weaker isolation level is – transactions that are interested only in reads, can carry on their work with older values, or even with dirty reads!
- Following are four transaction isolation levels in SQL-92-
  - Read Uncommitted
  - Read Committed
  - Repeatable Reads
  - Serializable [we already know this]
- They are in the order of increased isolation.
- Increased isolation also means increased computation overhead. Higher the isolation lesser will be concurrency, that is

    - delayed scheduling of transaction operation, or even
    - some are getting denial of execution
- Isolation-levels are specified for a transaction
    - Different transaction may execute at different isolation levels
    - Isolation variations are always in terms of READs; that is you may read something that is not as per serializability principles
    - 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 lesser Isolation Levels

Read Uncommitted

- OK reading uncommitted data
- Dirty reads may occur
  Example:

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

- If T2 runs at "Read UnCommitted" level, then it allows reading data from T1 even before it commits. If T1 decides to abort then report of T2 is not as per "serialization principles" - "DIRTY READ" is occurring.
- Even when T1 commits, T2 might read some old data before T1 updates and some updated that is also not yielding "Serializable" execution.

Read Committed

- Reads only committed data at the time of "reading" it.
- However "inconsistent read" might occur. For example a transaction reads a data item. Some other transaction writes X and commits. Now if reads X again then it sees different value of X.
- Note that such a transaction is not serializable. Check and see if it forms a cycle in precedence graph?

Inconsistent reads are also referred as "Non-Repeatable reads". Inconsistent reads may have problems in some cases. Observe behavior of following execution schedules.

Example-1

```
T1: begin transaction;
T2: begin transaction;
T2: select salary from employee where ssn = '101' ;  //50000
T1: update employee set salary = 45000 where ssn = '101';
T2: select salary from employee where ssn = '101' ;  //50000
T1: commit;
T2: select salary from employee where ssn = '101' ;  //45000
T2: commit;
```

Example-2:

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

- Problem: T2 gets average and max from different data set?
- Reason: NOT SERIALIZABLE; conflicting operations are in different order?

When we are allow this transaction T reading all committed data by simultaneous transactions! We get inconsistent values of a data item in a transaction

Some tricky example

```
T1:    begin transaction;
T1:    update employee set salary = 45000 where ssn = '101';
T2:    begin transaction;
T2:    update employee set salary =
       (select salary from employee where ssn = '101' )+3000 where ssn = '101';
T1:    commit;
T2:    commit;
```

Can you guess final update of salary? Old(before T1 and T2 began) + 3000; not equal to any serial schedule; non-serializable?

Look at the conflicting operations; w1(salary) and w2(salary) ; w1(salary) and r2(salary); due to low isolation level in T2, r2(salary) will execute before w1(salary).

You can simulate this in postgresql

1. open two sql windows; let us say w1 and w2
2. Type all commands of T1 in w1; and all commands of T2 in w2
3. Issue the commands in above sequence in respective windows!

Again cause of problem is NON-Repeatable READ

Repeatable Reads Isolation level

- Repeated read of a "data item" is consistent- of course we read only committed; and this level guarantees consistent reads.
- This repeatable read should be applicable to a granularity level of the context
- However repeatable read does not give repeatable read of a relation with respect to new inserted tuples! Consider example below

Example-2:

```
T1:   begin transaction;
T1:   INSERT INTO emp VALUES( … ); say adds 10 rwos
T2:   begin transaction;
T2:   set transaction isolation level repeatable read;
T2:   SELECT average(salary) FROM emp;
T1:   COMMIT
T2:   SELECT max(salary) FROM emp;
```

Isolation Level Summary

Isolation Level/Problem	Dirty Read	Non-repeatable reads	Phantom rows
Read Uncommitted	Y	Y	Y
Read Committed	N	Y	Y
Repeatable Read	N	N	Y
Serializable	N	N	N

- Standard default: Serializable
- Weaker isolation levels
  - Increased concurrency + decreased overhead = increased performance
  - Weaker consistency guarantees
  - Default isolation levels Some systems have default Repeatable Read
- Isolation level are for a transaction
  - Each transaction's reads must conform to its isolation level