Unit 5

Transaction Processing

Content:

1. Concepts in Transaction processing
2. Transaction and System concepts
3. Desirable properties of transactions
4. Serial, non-serial and Conflict schedules
5. Testing for conflict serializablility
6. Transaction support in SQL

1. Transaction Processing

- **A Transaction** : logical unit of database processing that includes one or more access operations (read -retrieval, write -insert or update, delete).
- **Transaction processing systems** are system with large databases and hundreds of concurrent users executing database transaction.
- Example : Airline reservation, Banking , Credit card processing, stock market, supermarket etc…

Single user Versus Multiuser Systems

- One criteria for classifying database system is according to number of users who can use the system concurrently.
- **Single-User System :** at most one user at a time can use the system.
  - Ex: ATM System
- **Multiuser System :** Many users can use the system concurrently.
  - Ex: airline reservation system, system in bank etc…

□ Single user are mostly restricted to personal computer

Multiprogramming & Interleaving

- **Concurrency**
  - **Multiprogramming :** Allows the computer to execute multiple programs at the same time.
  - **Interleaving :** keeps the CPU busy when a process requires an input or output operation, the CPU switched to execute another process rather than remaining idle during I/O time .
- Most of the theory concerning concurrency control in databases is developed in terms of interleaved concurrency.

Transactions, Read and Write Operations, and DBMS Buffers

- A **Transaction**:
  - Logical unit of database processing that includes one or more access operations (read -retrieval, write - insert or update, delete).
- All database access operations between **Begin Transaction** and **End Transaction** statements are considered one logical transaction.
- If the database operations in a transaction do not update the database but only retrieve data , the transaction is called a **read-only transaction.**

Basic Operation

- **A database** is a collection of named data items
- Basic operations are **read** and **write**
    - **read_item(X):** Reads a database item named X into a program variable. To simplify our notation, we assume that the program variable is also named X.
    - **write_item(X):** Writes the value of program variable X into the database item named X.

read_item(X)

- **read_item(X)** command includes the following steps:
  - Find the address of the disk block that contains item X.
  - Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
  - Copy item X from the buffer to the program variable named X.

write_item(X)

- **write_item(X)** command includes the following steps:
  - Find the address of the disk block that contains item X.
  - Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
  - Copy item X from the program variable named X into its correct location in the buffer.
  - Store the updated block from the buffer back to disk (either immediately or at some later point in time).

Example

- FIGURE 17.2 Two sample transactions:
  - (a) Transaction T1
  - (b) Transaction T2

(a) T_1

read_item (X);
X:=X-N;
write_item (X);
read_item (Y);
Y:=Y+N;
write_item (Y);

(b) T_2

read_item (X);
X:=X+M;
write_item (X);

Why concurrency control is needed?

- **The Lost Update Problem**
  - This occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database item incorrect.
- **The Temporary Update (or Dirty Read) Problem**
  - This occurs when one transaction updates a database item and then the transaction fails for some reason.
  - The updated item is accessed by another transaction before it is changed back to its original value.

Why concurrency control is needed?

- **The Incorrect Summary Problem**
  - If one transaction is calculating an aggregate summary function on a number of records while other transactions are updating some of these records, the aggregate function may calculate some values before they are updated and others after they are updated.

The Lost Update Problem

Time

T1	T2
Read_item(X);	
X=:X-N;	
	Read_item(X);
	X=:X+M;
Write_item(X);	
Read_item(Y);	
	Write_item(X);
Y=:Y+N;	
Write_item(Y);	

Problem : Item X has incorrect value because its updated by T1 is lost

The Temporary Update (or Dirty Read) Problem

	T1	T2
	Read_item(X); X=:X-N; write_item(X);	
Time		Read_item(X); X=:X+M; write_item(X);
	read_item(Y);	

Problem : T1 fails and must change the value of X back to its old value, meanwhile T2 has read the temporary value of x.

The Incorrect Summary Problem

Time

T1	T3
	Sum:=0;
	Read_item(A);
	sum:=sum+A;
Read_item(X);	.
X=:X-N;	.
write_item(X);	
	Read_item(X);
	sum:=sum+x;
	read_item(Y);
	sum:=sum+Y;
read_item(Y);	
Y:=Y+N;	
write_item(Y);	

Problem : T3 reads X after N is subtracted and reads Y before N is added; a wrong summery in a result.

Why recovery is needed?

- Types Of Failure:
  - **A computer failure (system crash):**
    - A hardware or software error occurs in the computer system during transaction execution. If the hardware crashes, the contents of the computer's internal memory may be lost.
  - **A transaction or system error:**
    - Some operation in the transaction may cause it to fail, such as integer overflow or division by zero.
    - Transaction failure may also occur because of erroneous parameter values or because of a logical programming error. In addition, the user may interrupt the transaction during its execution.

- **Local errors or exception conditions detected by the transaction:**
  - Certain conditions necessitate cancellation of the transaction.
  - For example, data for the transaction may not be found. A condition, such as insufficient account balance in a banking database, may cause a transaction, such as a fund withdrawal from that account, to be canceled.
  - A programmed abort in the transaction causes it to fail.
- **Concurrency control enforcement:**
  - The concurrency control method may decide to abort the transaction, to be restarted later, because it violates serializablility or because several transactions are in a state of deadlock.

- **Disk failure:**
  - Some disk blocks may lose their data because of a read or write malfunction or because of a disk read/write head crash.
  - This may happen during a read or a write operation of the transaction.
- **Physical problems and catastrophes:**
  - This refers to an endless list of problems that includes power or air-conditioning failure, fire, theft , overwriting disks or tapes by mistake, and mounting of a wrong tape by the operator.

2. Transaction and System concepts

- A **transaction** is an atomic unit of work that is either completed in its entirety or not done at all.
  - For recovery purposes, the system needs to keep track of when the transaction starts, terminates, and commits or aborts.

The recovery manager keeps track of the following operations :

- BEGIN_TRANSACTION
- READ OR WRITE
- COMMIT_TRANSACTION
- END_TRANSACTION
- ROLLBACK

The recovery manager keeps track of the following operations :

- **BEGIN_TRANSACTION**
  - This marks the beginning of transaction execution.
- **READ OR WRITE**
  - These specify read or write operations on the database items that are executed as part of a transaction.
- **COMMIT_TRANSACTION**
  - This signals a successful end of the transaction so that any changes (updates) executed by the transaction can be safely committed to the database and will not be undone.

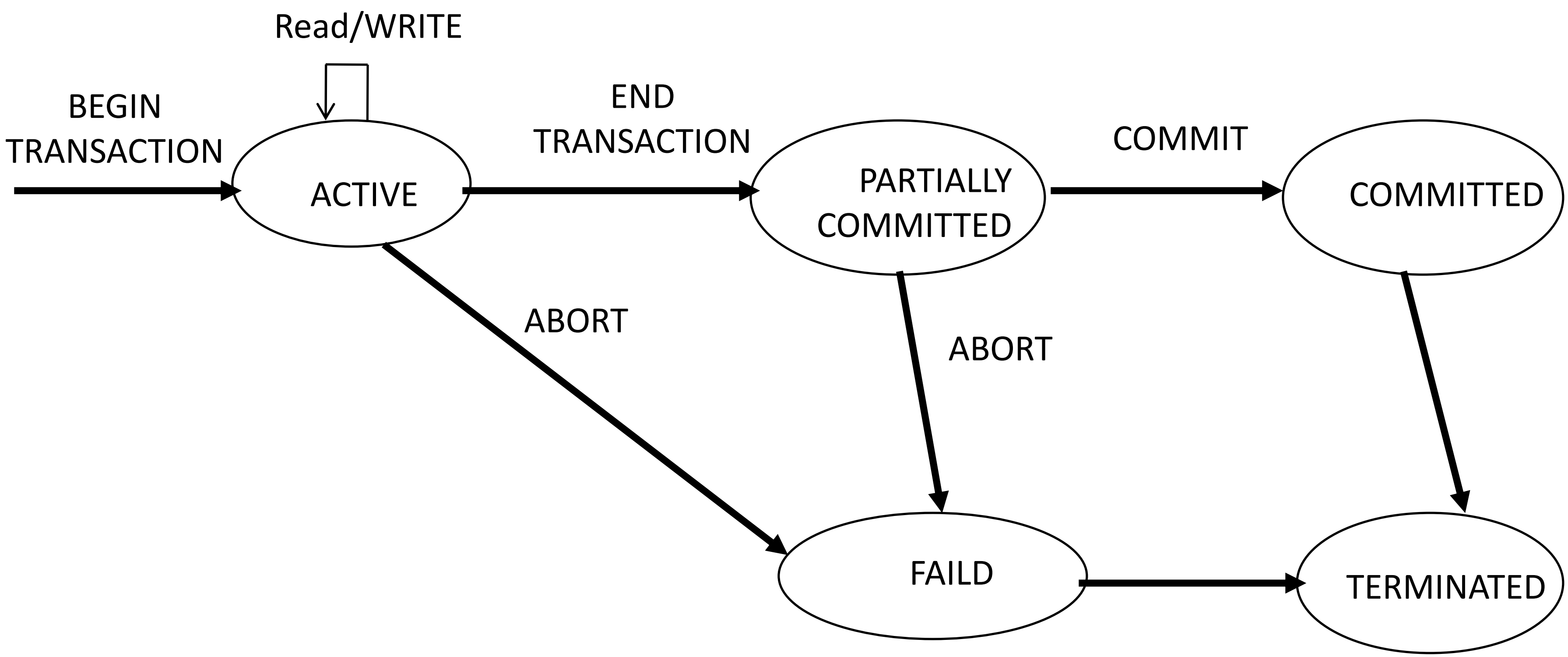
- **END_TRANSACTION**
  - This specifies that read and write transaction operations have ended and marks the end limit of transaction execution.
  - At this point it may be necessary to check
    - whether the changes introduced by the transaction can be permanently applied to the database
    - whether the transaction has to be aborted because it violates concurrency control or for some other reason.

- **ROLLBACK**
    - This signals that the transaction has ended unsuccessfully, so that any changes or effects that the transaction may have applied to the database must be undone.

Transaction states:

- Active state
- Partially committed state
- Committed state
- Failed state
- Terminated State

State Transition Diagram



State Transition Diagram illustrating the state for transaction Execution.

The System Log

- **Log or Journal :** The log keeps track of all transaction operations that affect the values of database items.
  - This information may be needed to permit recovery from transaction failures.
  - The log is kept on disk, so it is not affected by any type of failure except for disk or catastrophic failure.
  - In addition, the log is periodically backed up to archival storage (tape) to guard against such catastrophic failures.

- T refers to a unique **transaction-id** that is generated automatically by the system and is used to identify each transaction:
- **Types of log record:**
  - **[start_transaction,T]:** Indicates that transaction T has started execution.
  - **[write_item, T, X ,old_value ,new_value] :** Indicates that transaction T has changed the value of database item X from old value to new value. (new value may not be recorded)

- **[read_item, T, X]:** Indicates that transaction T has read the value of database item X.
- **[commit, T] :** Indicates that transaction T has completed successfully, committed (recorded permanently) to the database.
- **[abort, T] :** Indicates that transaction T has been aborted.

Recovery using log records:

- If the system crashes, we can recover to a consistent database state by examining the log by two technique.
    1. Because the log contains a record of every write operation that changes the value of some database item, it is possible to **undo** the effect of these write operations of a transaction T by tracing backward through the log and resetting all items changed by a write operation of T to their old values.

Recovery using log records:

2. We can also **redo** the effect of the write operations of a transaction T by tracing forward through the log and setting all items changed by a write operation of T (that did not get done permanently) to their new values.

Commit Point of a Transaction:

- **Definition a Commit Point:**
  - A transaction T reaches its **commit point** when all its operations that access the database have been executed successfully *and* the effect of all the transaction operations on the database has been recorded in the log.
  - Beyond the commit point, the transaction is said to be committed, and its effect is assumed to be permanently recorded in the database.
  - The transaction then writes an entry [commit, T] into the log.

- **Roll Back of transactions:**
  - Needed for transactions that have a [start_transaction, T] entry into the log but no commit entry [commit, T] into the log.

3. Desirable properties of transactions

- ACID should be enforced by the concurrency control and recovery methods of the DBMS.
- ACID properties of transactions :
  - Atomicity
  - Consistency Preservation
  - Isolation
  - Durability

- ***Atomicity*** **:** A transaction is an atomic unit of processing; it is either performed entirely or not performed at all.

  (It is the responsibility of recovery to ensure recovery)

- ***Consistency*** **:** A correct execution of the transaction must take the database from one consistent state to another.

  (It is the responsibility of the applications and DBMS or programmer to maintain the constraints)

- ***Isolation*** **:** A transaction should not make its updates visible to other transactions until it is committed.
  - This property, when enforced strictly, solves the temporary update problem and eliminates cascading rollbacks of transactions.

(It is the responsibility of concurrency)

- ***<u>Durability</u>*** **:** Once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failure.
    - i.e. those changes must not be lost because of any failure.

(It is the responsibility of recovery)

4. Schedule of Transaction

- **Transaction schedule or history**:
  - When transactions are executing concurrently in an interleaved fashion, the order of execution of operations from the various transactions is known as a transaction schedule (or history).
- A **schedule** (or **history**) S of n transactions T1, T2, ..., Tn:
  - It is an ordering of the operations of the transactions subject to the constraint that, for each transaction Ti that participates in S, the operations of T1 in S must appear in the same order in which they occur in T1.

Serial schedules

- A schedule S is **serial** if, for every transaction T participating in the schedule, all the operations of T are executed consecutively in the schedule.

- Only one transaction at a time is active.
- The commit(or abort) of the active transaction initiate execution of next transaction.
- No interleaving occurs in serial schedule.

- if we consider transactions to be independent, then every serial schedule is considered correct.
- For example, if T1 and T2 are the participating transactions in S, then T1followed by T2 or T2 followed by T1 are called as serial schedules
- The schedules that do not follow this consecutiveness are called as non-serial schedules

Problems of serial schedules :

- They limit concurrency or interleaving of operations
  - if a transaction waits for an I/O operation to complete, we cannot switch the CPU Processor to another transaction.
  - if some transaction T is long , the other transactions must wait for T to complete all its operations.

Non - Serial schedules

- A schedule S is **non-serial** if, transaction occur in the interleaved manner.

- Transactions is to be dependent with each other then it said to be non-serial schedule.

Serial and Non Serial Schedule

T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

Diagram (a) : Serial Schedule

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

Diagram (b) : Non-Serial Schedule

Conflict Schedule

- **Two operations are said to be conflict if:**
  - They belong to different transactions.
  - They access the same item X.
  - At least one the operations is a write_item(X).

Ex:

S1: $r_1(x);r_2(x);w_1(x);r_1(y);w_2(x);w_1(y);$

conflicts: $[r_1(x);w_2(x)] – [r_2(x);w_1(x)] – [w_1(x); w_2(x)]$

- Instructions I_i and I_J of transactions T_i and T_J respectively, conflict if and only if there exists some item Q accessed by both I_i and I_j, and at least one of these instructions wrote Q.
  - I_i = read(Q), I_J = read(Q). I_i and I_J don't conflict.
  - I_i = read(Q), I_J = write(Q). They conflict.
  - I_i = write(Q), I_J= read(Q). They conflict
  - I_i = write(Q), I_J = write(Q). They conflict

Serializable Schedule

- A schedule S is **Serializable** , if it is *equivalent to* some serial schedule of the same set of participating *committed transactions in S.*
  - Conflict equivalence
  - Conflict Serializable
- Note: The difference between a serial and Serializable schedule is that a serial schedule is not interleaved, but Serializable schedule is interleaved.

Terms

- **Serializable schedule:**
  - A schedule S is Serializable if it is equivalent to some serial schedule of the same n transactions
- **Result equivalent:**
  - Two schedules are called result equivalent if they produce the same final state of the database.

- **Conflict equivalent:**
  - Two schedules are said to be conflict equivalent if the order of any two conflicting operations is the same in both schedules.
- **Conflict Serializable :**
  - A schedule S is said to be conflict Serializable if it is conflict equivalent to some serial schedule S’.

Testing for Conflict Serializablility of a Schedule (Precedence Graph)

- **Algorithm for Testing conflict serializablility of schedule S.**

1. For each transaction T_i participating in schedule S, create a node labeled T_i in the precedence graph.
2. For each case in S where T_j executes a read_item(X) after T_i executes a write_item(X), create an edge $(T_i \rightarrow T_j)$ in the precedence graph.
3. For each case in S where T_j executes a write_item(X) after T_i executes a read_item(X), create an edge $(T_i \rightarrow T_j)$ in the precedence graph.
4. For each case in S where T_j executes a write_item(X) after T_i executes a write_item(X), create an edge $(T_i \rightarrow T_j)$ in the precedence graph.
5. The schedule S is serializable if and only if the precedence graph has no cycles.

(b)

Time

T_1	T_2
read_item(X);	
$X := X - N$;	
write_item(X);	
read_item(Y);	
$Y := Y + N$;	
write_item(Y);	
	read_item(X);
	$X := X + M$;
	write_item(X);

Schedule A

Time

T_1	T_2
	read_item(X);
	$X := X + M$;
	write_item(X);
read_item(X);	
$X := X - N$;	
write_item(X);	
read_item(Y);	
$Y := Y + N$;	
write_item(Y);	

Schedule B

Time

T_1	T_2
read_item(X);	
$X := X - N$;	
	read_item(X);
	$X := X + M$;
write_item(X);	
read_item(Y);	
	write_item(X);
$Y := Y + N$;	
write_item(Y);	

Schedule C

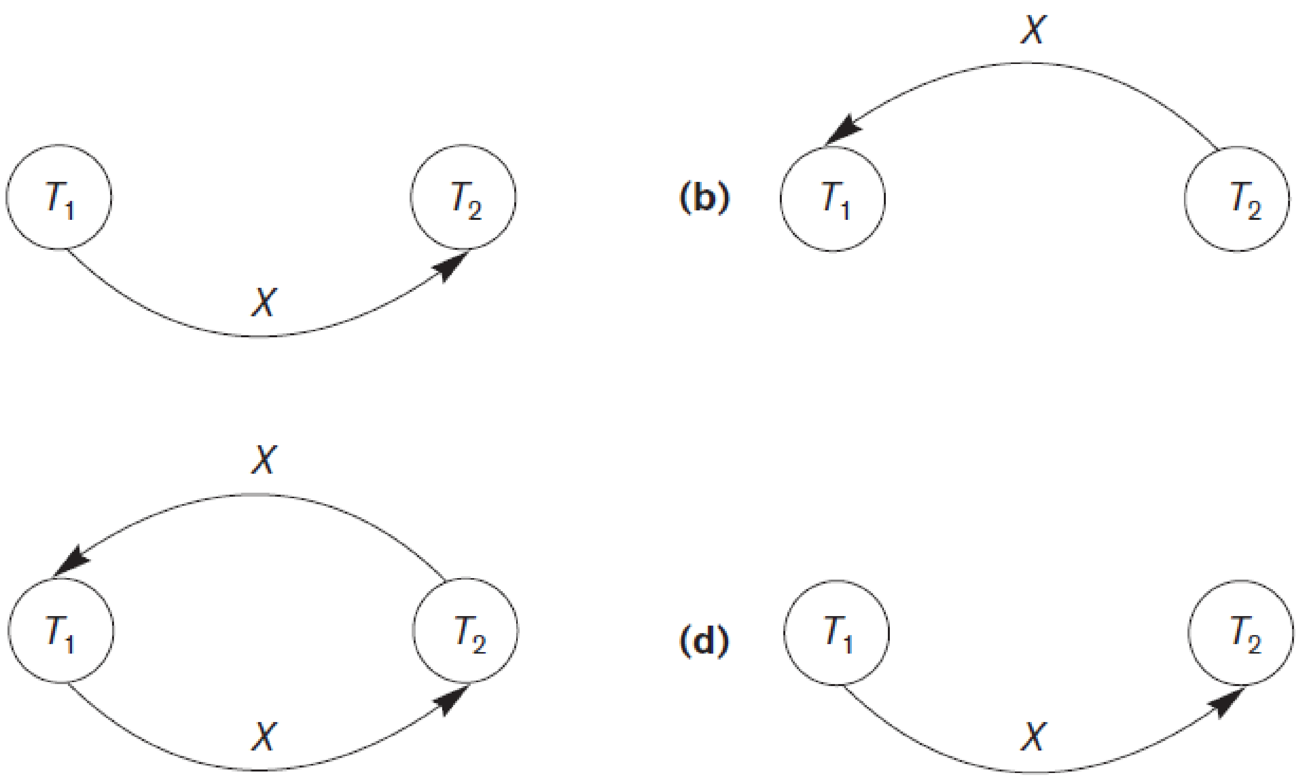
Time

T_1	T_2
read_item(X);	
$X := X - N$;	
write_item(X);	
	read_item(X);
	$X := X + M$;
	write_item(X);
read_item(Y);	
$Y := Y + N$;	
write_item(Y);	

Schedule D

Testing for Conflict Serializablility of a Schedule (Precedence Graph)

- [Algorithm](#) for Testing conflict serializablility of schedule S.



6. Transaction Support in SQL

- A **single** SQL statement is always considered to be **atomic**.
  - Either the statement completes execution without error or it fails and leaves the database unchanged.
- With SQL, there is no explicit Begin Transaction statement.
  - Transaction initiation is done implicitly when particular SQL statements are encountered.
- Every transaction must have an explicit end statement, which is either a COMMIT or ROLLBACK.

Characteristics Attributes of Transaction

Characteristics specified by a SET TRANSACTION statement in SQL:

- **Access mode:**
  - READ ONLY or READ WRITE.
    - The default is READ WRITE unless the isolation level of READ UNCOMITTED is specified, in which case READ ONLY is assumed.
- **Diagnostic area size**
  - Diagnostic size n, specifies an integer value n, indicating the number of conditions that can be held simultaneously in the diagnostic area.

- **Isolation level**
  - it is specified using statement ISOLATION LEVEL <isolation>,
    - where <isolation> can be READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ or SERIALIZABLE. The default is SERIALIZABLE.
    - However, if any transaction executes at a lower level, then serializablility may be violated.

Potential problem with lower isolation levels:

- **Dirty Read**:
  - Reading a value that was written by a transaction which failed.
- **Non-repeatable Read (incorrect – summery Problem)**
  - Allowing another transaction to write a new value between multiple reads of one transaction.
  - A transaction T1 may read a given value from a table. If another transaction T2 later updates that value and T1 reads that value again, T1 will see a different value.

- **Phantoms:**
  - New rows being read using the same read with a condition.
    - A transaction T1 may read a set of rows from a table, perhaps based on some condition specified in the SQL WHERE clause.
    - Now suppose that a transaction T2 inserts a new row that also satisfies the WHERE clause condition of T1, into the table used by T1.
    - If T1 is repeated, then T1 will see a row that previously did not exist, called a phantom.

Possible violation of serializablility:

Isolation Level	Dirty Read	Non-Repeatable Read	Phantom
Read Uncommitted	Yes	Yes	Yes
Read Committed	No	Yes	Yes
Repeatable Read	No	No	Yes
Serializablility	No	No	No