INTERLEAVING:

Since the DBMS is a centralized system that will service several clients, it is very possible that several clients will want to execute instructions on the database "at the same time"

This will require the DBMS to "decide" how to proceed with those requests. There are two choices:

1) The requests are processed sequentially in the order in which they arrived.
2) Interleaved: The requests are processed concurrently, breaking the requests into smaller time-manageable pieces and devoting some time to each one of those pieces. Very much like an Operating System works with concurrent processes.

Suppose that there are two requests:

- The first one is from a user in an ATM machine that wants to withdraw money
- The second is a big company that will run the entire payroll for their thousands of employees.

If the DBMS processes requests sequentially, and the second request is processed first, the poor guy at the ATM machine will have to wait for a long time to get his money!

ADVANTAGES OF INTERLEAVING:
- The performance of the system is better if the CPU is working on several transactions simultaneously, since disk storage is slower.
- Users should not have to wait for larger transactions to compute first.

WHAT DOES A TRANSACTION LOOK LIKE?

Let us assume then that we will grant requests by interleaving them.

Example:

Suppose that we want to transfer 100 dollars from account A to account B.
The operations that are requested are:
A=A-100
B=B+100
Where A and B represent the storage location of the balance of accounts A and B respectively.

From the perspective of the client (user) the operations that are requested from the DBMS would look as follows:

- Read balance from account A into variable a
- a=a-100
- Write balance a into account A
- Read balance from account B into variable b
- b=b+100
- Write balance b into account B

TRANSACTIONS

Informally a transaction is a logical unit of work

What does the DBMS see?

R(A)	Read balance from account A into variable A
	A=A-100
W(A)	Write balance a into account A
R(B)	Read balance from account B into variable B
	B=B+100
W(B)	Write balance b into account B

The DBMS will only see a sequence of Read R() and write W() operations

For the sake of simplicity:

We use the notation R(A) to say that we read from an element A in the database into a variable A in the client.

The notation W(A) means that we take the variable A in the client and write its value into element A in the database.

So the sequence: R(A);A=A-100;W(A)
Withdraws 100 from account A

PROBLEMS WITH INTERLEAVING

Alice and Bob show up at different ATM machines at the same time.
They have a joint checking account with a current balance of $60
Alice wants to withdraw $40
Bob wants to withdraw $50

The operations to do that should look like this:

Alice:	Bob:
R(A)	R(A)
A=A-40	A=A-50
W(A)	W(A)

The ATM machine has no problem processing a withdrawal if there is enough balance in the account.

Using interleaving, the DBMS might see the following operations:

Alice:	R(A)		W(A-40)	
Bob:		R(A)		W(A-50)

For convenience we write them in such a way that time increases to the right.
Since the initial balance was $60, the final balance after Bob's and Alice's operations have been executed will be $10! (Withdrawal of $90 from an account with $60, and still $10 remaining).

TRANSACTIONS

A solution to the previous problem is to force several operations on the database to be considered indivisible, in such a way that it will look as if they were executed as a unit.

Formally: A **transaction** is a sequence of R,W operations that must be processed as a unit or not at all.

A transaction is usually started by a “*begin transaction”* command, and is terminated by a “*commit*” or “*Abort*” command.

The *Abort* command must Roll back the operations performed so far.

The effects of a transaction become permanent only after it has *Committed.*

TRANSACTIONS (cont’d)

```
Consistent state  ______________________
                   begin transaction
                   …
                   …
                   …
                   …
                   commit
Consistent state  ______________________
```

We will assume that the Database is in a consistent state immediately before a transaction is executed, and will be in a consistent state once the transaction has committed.

IMPORTANT: A transaction is not required to keep the database in a consistent state while it is being executed

A consistent state is a state that:

- Satisfies the integrity constraints described by the schema.
- Satisfies other constraints required by the system that might not be explicitly stated as integrity constraints.

ACID PROPERTIES

The following properties ensure database reliability:

- **A**tomicity: A transaction must be either: 1) executed entirely or 2) not executed at all. It should not be possible to execute one part of a transaction and not another.

- **C**onsitency: If we assume that the database is in a consistent state before the transaction is executed, then the database must remain in a consistent state after the transaction has committed.

- **I**solation: Transactions executed concurrently must not interact. A transaction cannot exchange messages with another transaction. Each transaction must "feel" as if it is the only transaction being executed at a time. Interaction between transactions negates that condition.

- **D**urability: The effects of a transaction become permanent after it has committed.

ACID properties also deal with other problems besides concurrency:

- System crashes, power failures, etc.
- Suspend integrity constraint checking in while Xacts are being processed. (A(foreign key (person,spouse) references A(spouse,person))

WHO IS RESPONSIBLE FOR THE ENFORCEMENT OF ACID PROPERTIES?

Atomicity: The ***DBMS*** must make sure that a given transaction is executed “all or nothing”. If there are problems while executing a transaction, and the transaction cannot be entirely executed, it must be aborted and its effects rolled-back.

Consistency: The **programmer** is responsible for writing transactions in such a way that after their execution, the database remains in a consistent state. Notice that not only those constraints given in the schema determine consistency (example: a transaction that is supposed to transfer balance from one account to another, but that does not correctly compute the balance to be written on one of the accounts, would still satisfy integrity constraints).

Isolation: The **DBMS** will provide the programmer with the choice of several levels of isolation (when to make changes accessible to other transactions). The concurrency control of the DBMS will enforce the desired level of isolation. Depending on the level of isolation, different types of problems (anomalies) might occur. The **programmer** must not allow transactions to exchange messages

Durability: The ***DBMS*** must make sure that the effects of a transaction are permanent even in the case of system failure.

SCHEDULES

A schedule is a description of order in which interleaved transactions are executed.

They usually use the notation described before (when talking about transactions) where time is represented as increasing to the right.

The following is an example of a schedule:

```
T1:  R(A)            W(A-40)
T2:         R(A)                W(A-50)
```

Other operations that can be included in a transaction are: **COMMIT** and **ABORT**

ANOMALIES PRODUCED BY INTERLEAVING TRANSACTIONS

We will study possible anomalies that occur because of interleaving transactions so that we can devise a method to reduce them.

1) Reading Uncommitted Data (Dirty Read)

EXAMPLE:
Analyze the following schedule where:

- T1: transfers 1,000,000 dollars from account A to account B.
- T2: deposits 100 dollars into account A

```
T1:  R(A)W(A-1,000,000)                                R(B)W(B+1,000,000) Abort
T2:                       R(A)W(A+100) Commit
```

Notice that transaction T2 will read A once 1,000,000 have been subtracted from the account and write that amount + 100. Since it commits, the changes must be permanent because of **DURABILITY**. However, transaction T1 aborts because (whatever reason) and should roll back! Did someone loose money?

In general, the pattern for a dirty read is:

```
T1: R(A)W(A)                                R(B)W(B) Abort
T2:                  R(A)W(A) Commit
```

2) Unrepeatable Read

EXAMPLE:
Analyze the following schedule where:

- T1: Checks to see if room A is available at a hotel. Then, checks again before reserving it.
- T2: Checks to see if room A is available at the same hotel. Then reserves it and commits.

T1:	**R(A)**		**R(A) Commit**
T2:		**R(A)W(A) Commit**	

Notice that the second time that T1 checks the availability of the room, it will not get the same result as the first time that it did so!

In general, the pattern for an unrepeatable read is the same as above:

T1:	**R(A)**		**R(A) Commit**
T2:		**R(A)W(A) Commit**	

3) Overwriting Committed Data

T1:	**W(A)**		**W(B) Commit**
T2:		**W(A) W(B) Commit**	

4) Phantom Read

Between two Read operations insert/delete operations might change the contents of the table.

Example:

```
T1:  select count(*)                                    select count(*)
     from students;                                     from students;
T2:                      insert into students (id)
                         values (150003874);
```

Notice that the number obtained from count(*) in T1 is not the same both times that it is executed. Remember **ISOLATION**?

The Phantom Read Anomaly does not depend only on R/W, but on other Db operations such as insert/delete

TYPES OF SCHEDULES

- Serial schedule: One transaction must be completely processed before starting the execution of another transaction.
- Equivalent schedules: two schedules such that their final effect on the objects of the database is the same.
- Serializable schedule: A schedule such that it is equivalent to some Serial schedule.

SERIAL SCHEDULES

Suppose that two transactions will run concurrently. The first one, transfers $100 from account A to account B. The second one, adds 5% interest to both accounts. Suppose that both accounts start with a balance of $300

```
T1:  R(A)W(A+100)R(B)W(B-100)
T2:  R(A)W(A*1.05)R(B)W(B*1.05)
```

In a ***serial schedule*** we execute one transaction at a time, and do not start working on another transaction until the current transaction has been completely executed.

In our example, we have two possible serial schedules for T1 and T2:

Case 1) Execute T1 first

```
T1:  R(A)W(A+100)R(B)W(B-100)
T2:                              R(A)W(A*1.05)R(B)W(B*1.05)
```

Case 2) Execute T2 first

```
T1:                              R(A)W(A+100)R(B)W(B-100)
T2:  R(A)W(A*1.05)R(B)W(B*1.05)
```

Case 1) Execute T1 first

```
T1:  R(A)W(A+100)R(B)W(B-100)
T2:                                         R(A)W(A*1.05)R(B)W(B*1.05)
```

	Execution:	In the database A	B	
		300	300	
T1	R(A)	300	300	
	W(A+100)	400	300	← Notice inconsistent state!
	R(B)	400	300	
	W(B-100)	400	200	
T2	R(A)	400	200	
	W(A*1.05)	420	200	
	R(B)	420	200	
	W(B*1.05)	$\boxed{420}$	$\boxed{210}$	

Case 2) Execute T2 first

```
T1:  R(A)W(A+100)R(B)W(B-100)
T2:                            R(A)W(A*1.05)R(B)W(B*1.05)
```

	Execution:	In the database A	B
		300	300
T2	R(A)	300	300
	W(A*1.05)	315	300
	R(B)	315	300
	W(B*1.05)	315	315
T1	R(A)	315	315
	W(A+100)	415	315
	R(B)	415	315
	W(B-100)	$\boxed{415}$	$\boxed{215}$

SERIAL SCHEDULES (cont'd)

Even though both schedules [T1 T2] and [T2 T1] satisfy the definition of a *Serial Schedule*, the final effect on the database is not the same:

T1 T2	A=420	B=210
T2 T1	A=415	B=215

We have no reason to prefer one schedule over another. Why should executing T1 first would be better (worse) than executing T2 first?

In both cases the transactions "feel" as if they were the only ones being executed.

Therefore, both schedules are OK and are considered to be Serial Schedules.

Conclusion: Serial Schedules are not unique.

Goal: be able to interleave transactions in such a way that the effect on the database is the same as the effect of some serial schedule!

ACID Implementation

As we said before, the DBMS is responsible for:

- Atomicity: Make sure that ALL or NOTHING. Means that it must provide the mechanism to Rollback a transaction.
- Consistency: Detects the violation of integrity constraints and provides methods of dealing with them (ON DELETE CASCADE, etc.). Suspend checking of constraints int the middle of Xacts.
- Isolation: Will provide several levels of isolation to prevent some or all anomalies.
- Durability: Store to disk, make backups, etc.

IMPLEMENTATION OF ISOLATION

Use of two types of locks:

S-lock: Shared lock. Is needed before reading data
X-lock: Exclusive lock. Is needed before writing data.

Granularity of locks: Locks can be requested/released for:

- Single tuple
- Range of tuples
- Entire table

Rules:

1) If a Xact holds an X-lock on A, no other transaction can get an S-lock or an X-lock on that object.

2) If a Xact holds a S-lock on A, another transaction can get an S-lock on A, but not an X-lock on A.

A TRANSACTION MIGHT HAVE TO WAIT TO GET THE REQUIRED LOCK

This might cause a *deadlock* to occur.

T1:	X-lock(A)		W(A)		X-lock(B)		W(B)	
T2:		X-lock(B)		W(B)		X-lock(A)		W(A)

The DBMS must detect the deadlock and restart one of the Xacts:

- Precedence graphs
- Time stamps

ISOLATION LEVELS

When to release the locks?

- **X-locks** held by a Xact are released after the Xact has committed.
- In the case of **S-locks** held by a Xact there are three choices:
    1) Do not use S-locks
    2) They are released immediately after reading the object.
    3) They are released after the Xact has committed.

4) Read uncommitted

No S-locks are used.
Allows all of the anomalies described before

3) Read committed

S-lock is needed to Read and released after each Read.

This was our “Dirty Read” pattern:

~~**T1: R(A)W(A)   R(B)W(B) Abort**~~
~~**T2:   R(A)W(A) Commit**~~

This can no longer happen T2 will no longer be able to read the uncommitted data A because it will request a S-lock and will have to wait until T1 commits to get it.

To think about: It allows unrepeatable reads!

2) Repeatable Read

S-lock is needed to Read and released after the Xact has committed.

This was our “Unrepeatable Read” pattern:

~~**T1: R(A) R(A) Commit**~~
~~**T2: R(A)W(A) Commit**~~

This can no longer happen T2 will no longer be able to write A because it will need an X-lock(A) and will have to wait until T1 commits to get it.

To think about: It allows phantom reads!

1) Serializable

- All locks acquired are kept until the Xact has committed.
- Lock indexes used by the query (holding off inserts/deletes)

STRICT 2PL: All locks released only after transaction has committed

Two phase locking allows only serial schedules.

Summary:

Isolation level	Dirty Read	Unrepeatable Read	Phantom Read
Read Uncommitted	May occur	May occur	May occur
Read Committed		May occur	May occur
Repeatable Read			May occur
Serializable			

TRANSACTIONS IN MYSQL

```
START TRANSACTION
.
.
.
COMMIT
```

```
START TRANSACTION
.
.
.
ROLLBACK
```

```
SET (GLOBAL | SESSION) TRANSACTION ISOLATION LEVEL
(READ UNCOMMITED | READ COMMITED | REPEATABLE READ |
 SERIALIZABLE)
```

If no GLOBAL or SESSION is given, then it will only affect the next Xact

ATOMICITY IMPLEMENTATION

Logs are used to deal with aborting transactions and system crashes

The Log has: Old value and New value
The Log must be stored (disk) before the actual changed data (WAL - Write Ahead Logging)

When a transaction commits/aborts, a log record must indicate this action.