MULTIUSER DATABASES: Concurrency and Transaction Management
Banking Application

- Entities in a banking application:
  - Customers
  - Employees
  - Accounts
- In an operational bank database, customers use the ATMs, internet, and phones to interact with their accounts
- This is a multiuser database since many customers may be connected to the bank database and doing money transfers, checking their balance etc.
Banking Application

- Consider that CM is transferring 100YTL from his account to BT’s account.
- The following operations take place:
  - Read the amount of money in the account of CM (a)
    - a := a – 100
  - Read the amount of money in BT’s account (r)
    - r = r + 100
- At the same time, the bank calculates the total amount of money stored in the accounts
  - Read amount of money in the accounts one by one
  - Add the amounts to the sum.
Banking Application

CM

400 YTL

BT

100 YTL
Banking Application

300 YTL  →  100 YTL

100 YTL
Banking Application

300 YTL

200 YTL
Banking Application

300 YTL

200 YTL

Sum

0
Banking Application

Sum := sum + 300

300 YTL

200 YTL

300
Banking Application

Sum := sum + 200

300 YTL

200 YTL

500
Banking Application

Things are fine if I finish the money transfer and then calculate the sum. But consider the following case
Banking Application

300 YTL

100 YTL

sum

0
Banking Application

300 YTL

100 YTL

Sum := sum + 300

300
Banking Application

300 YTL

100 YTL

Sum := sum + 100

400
Banking Application

300 YTL → 100 YTL → 200 YTL → sum 400 YTL
Concurrency

- Interleaving the execution of the operations such as the money transfer and account sum.
- Concurrency is needed for performance reasons (ex: using the CPU when somebody else is accessing the disk)
Concurrency

- A user program may be doing many different operations but from a database point of view, only R/W operations are of interest.

- A transaction is the DBMS’s abstract view of a user program: a sequence of reads and writes performed as a single logical unit of work
  - Ex: **Transaction1**: R(Account1), Read(Account2), Write(Account1)
Concurrency in a DBMS

- Users submit transactions, and can think of each transaction as executing by itself.
  - Concurrency is achieved by the DBMS, which interleaves actions (reads/writes of DB objects) of various transactions.
  - Each transaction must leave the database in a consistent state if the DB is consistent when the transaction begins.
Concurrency in a DBMS

- DBMS will enforce some ICs, depending on the ICs declared in CREATE TABLE statements.
- Beyond this, the DBMS does not really understand the semantics of the data. (e.g., it does not understand how the interest on a bank account is computed).

**Main Issues:** Effect of *interleaving* transactions, and *crashes.*
Multiuser centralized transaction processing system.

Databases and Transaction Processing  Lewis, Bernstein, Kifer
Two-tiered multiuser distributed transaction processing system.

Databases and Transaction Processing  (Lewis, Bernstein, Kifer)
Three-tiered multiuser distributed transaction processing system.

Databases and Transaction Processing  (Lewis, Bernstein, Kifer)
ACID Properties of transactions

- Atomicity
- Consistency
- Isolation
- Durability
Atomicity of Transactions

- A transaction might *commit* after completing all its actions, or it could *abort* (or be aborted by the DBMS) after executing some actions.
Atomicity of Transactions

A very important property guaranteed by the DBMS for all transactions is that they are \textit{atomic}. That is, a user can think of a transaction as always executing all its actions in one step, or not executing any actions at all.

- DBMS \textit{logs} all actions so that it can \textit{undo} the actions of aborted transactions.
Transactions in SQL

UPDATE authors
SET au_fname = 'John'
WHERE au_id = '172-32-1176'

This is an auto-commit transaction with only one statement

REF: http://www.sqlteam.com/article/introduction-to-transactions
BEGIN TRAN

UPDATE authors
SET au_fname = 'John'
WHERE au_id = '172-32-1176'

UPDATE authors
SET au_fname = 'Marg'
WHERE au_id = '213-46-8915'

COMMIT TRAN

REF: http://www.sqlteam.com/article/introduction-to-transactions
Transactions in SQL

BEGIN TRAN

UPDATE authors
SET au_fname = 'John'
WHERE au_id = '172-32-1176'

UPDATE authors
SET au_fname = 'Marg'
WHERE au_id = '213-46-8915'

COMMIT TRAN

REF: http://www.sqlteam.com/article/introduction-to-transactions
Example

Consider two transactions

T1: \[ \text{BEGIN } A = A + 100, \ B = B - 100 \ \text{END} \]

T2: \[ \text{BEGIN } A = 1.06 \times A, \ B = 1.06 \times B \ \text{END} \]

Intuitively, the first transaction is transferring $100 from B’s account to A’s account. The second is crediting both accounts with a 6% interest payment.

There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together. However, the net effect must be equivalent to these two transactions running serially in some order.
Example (Contd.)

- Consider a possible interleaving (*schedule*):

<table>
<thead>
<tr>
<th></th>
<th>T1:</th>
<th>T2:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A=A+100,</td>
<td>B=B-100</td>
</tr>
<tr>
<td></td>
<td>A=1.06*A,</td>
<td>B=1.06*B</td>
</tr>
</tbody>
</table>

- This is OK. But what about:

<table>
<thead>
<tr>
<th></th>
<th>T1:</th>
<th>T2:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A=A+100,</td>
<td>B=B-100</td>
</tr>
<tr>
<td></td>
<td>A=1.06*A,</td>
<td>B=1.06*B</td>
</tr>
</tbody>
</table>

- The DBMS’s view of the second schedule:

<table>
<thead>
<tr>
<th></th>
<th>T1:</th>
<th>T2:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R(A), W(A),</td>
<td>R(B), W(B)</td>
</tr>
<tr>
<td></td>
<td>R(A), W(A),</td>
<td>R(B), W(B)</td>
</tr>
</tbody>
</table>
Scheduling Transactions

- **Serial schedule**: Schedule that does not interleave the actions of different transactions.

```
T1: A = A + 100, B = B - 100
T2: A = 1.06 * A, B = 1.06 * B
```
Scheduling Transactions

- **Equivalent schedules.**
  - Schedules involving the same set of operations on the same data objects

Schedule 1

<table>
<thead>
<tr>
<th>T1: R(A), W(A), R(B), W(B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2: R(A), W(A)</td>
</tr>
</tbody>
</table>

Schedule 2

<table>
<thead>
<tr>
<th>T1: R(A), W(A), R(B), W(B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2: R(A), W(A)</td>
</tr>
</tbody>
</table>
Scheduling Transactions

- **Equivalent schedules.**
  - Schedules with the same set of operations on the same data objects.
  - And, for any database state, the effect (on the set of objects in the database) of executing the first schedule is identical to the effect of executing the second schedule.

\[ DB' = DB'' \]
Scheduling Transactions

- **Serializable schedule**: A schedule that is equivalent to some serial execution of the transactions.

<table>
<thead>
<tr>
<th>Schedule 1</th>
<th>Serial Schedule A</th>
<th>Serial Schedule B</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1: R(A), W(A), R(B), W(B)</td>
<td>T1: R(A), W(A), R(B), W(B)</td>
<td>T1: R(A), W(A), R(B), W(B)</td>
</tr>
<tr>
<td>T2: R(A), W(A)</td>
<td>T2: R(A), W(A)</td>
<td>T2: R(A), W(A)</td>
</tr>
</tbody>
</table>

**Question:**
Is schedule 1 Equivalent to serial schedule A or B?
Scheduling Transactions

- If each transaction preserves consistency, every *serializable* schedule preserves consistency!
Anomalies with Interleaved Execution

- Reading Uncommitted Data (WR Conflicts, “dirty reads”):

<table>
<thead>
<tr>
<th>T1:</th>
<th>R(A), W(A), R(B), W(B), Abort</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2:</td>
<td>R(A), W(A), C</td>
</tr>
</tbody>
</table>

- What happens when T1 aborts?
Anomalies with Interleaved Execution

- Unrepeatable Reads (RW Conflicts):

  T1: R(A), R(A), R(A), W(A), C
  T2: R(A), W(A), C
Anomalies (Continued)

- Overwriting Uncommitted Data (WW Conflicts):

<table>
<thead>
<tr>
<th>T1:</th>
<th>W(A), W(B), C</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2:</td>
<td>W(A), W(B), C</td>
</tr>
</tbody>
</table>
Role of a concurrency control in a database system.

Databases and Transaction Processing  (Lewis, Bernstein, Kifer)
Lock-Based Concurrency Control

- Each transaction must obtain a **S (shared) lock** on object before reading, and an **X (exclusive) lock** on object before writing.
- An S or X lock is released when the corresponding object is no longer needed.
  - Ex: T1: S(A), R(A), Release_S(A), X(B), W(B), Release_X(B) ...
Lock-Based Concurrency Control

- X conflicts with X and S
  - No transaction can obtain an X lock on an object if some other transaction has an X or S lock on that object.
  - No transaction can obtain an S lock on an object if some other transaction has an X lock on that object
- S locks do not conflict with each other
  - Multiple transactions may obtain an S lock on the same object
Lock-Based Concurrency Control

- **Strict Two-phase Locking (Strict 2PL) Protocol:**
  - Each transaction must obtain a *S (shared)* lock on object before reading, and an *X (exclusive)* lock on object before writing.
  - All locks held by a transaction are released when the transaction completes.
  - If a transaction holds an X lock on an object, no other transaction can get a lock (S or X) on that object.
- Strict 2PL allows only **serializable** schedules.
Aborting a Transaction

- If a transaction $T_i$ is aborted,
  - all its actions have to be undone.
  - if $T_j$ reads an object last written by $T_i$, $T_j$ must be aborted as well! (called **cascading aborts**)
Aborting a Transaction

- Most systems avoid *cascading aborts* by releasing a transaction’s locks only at commit time.
  - If $T_i$ writes an object, $T_j$ can read this only after $T_i$ commits.
- In order to *undo* the actions of an aborted transaction, the DBMS maintains a *log* in which every write is recorded.
- Log is also used to recover from system crashes: all active transactions at the time of the crash are aborted when the system comes back up.
The Log

- The following actions are recorded in the log:
  - *Ti writes an object:* the old value and the new value.
    - Log record must go to disk *before* the changed page!
  - *Ti commits/aborts:* a log record indicating this action.

- Log records are chained together by transaction id, so it’s easy to undo a specific transaction.

- Log is often *duplexed* and *archived* on stable storage.

- All log related activities (and in fact, all CC related activities such as lock/unlock, dealing with deadlocks etc.) are handled transparently by the DBMS.
Recovering From a Crash

- There are 3 phases in the *Aries* recovery algorithm:
  - **Analysis**: Scan the log forward (from the most recent *checkpoint*) to identify all Xacts that were active, and all dirty pages in the buffer pool at the time of the crash.
  - **Redo**: Redoes all updates to dirty pages in the buffer pool, as needed, to ensure that all logged updates are in fact carried out and written to disk.
  - **Undo**: The writes of all Xacts that were active at the crash are undone (by restoring the *before value* of the update, which is in the log record for the update), working backwards in the log. (Some care must be taken to handle the case of a crash occurring during the recovery process!)
Conflict Serializable Schedules

- Two schedules are conflict equivalent if:
  - Involve the same actions of the same transactions
  - Every pair of conflicting actions is ordered the same way

**Schedule 1**

<table>
<thead>
<tr>
<th>T1:</th>
<th>R(A), W(A),</th>
<th>R(B), W(B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2:</td>
<td>R(A), W(A)</td>
<td>R(B)</td>
</tr>
</tbody>
</table>

**Schedule 2**

<table>
<thead>
<tr>
<th>T1:</th>
<th>R(A), W(A),</th>
<th>R(B), W(B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2:</td>
<td>R(A),</td>
<td>W(A) R(B)</td>
</tr>
</tbody>
</table>

Is schedule1 conflict equivalent to schedule2?
Conflict Serializable Schedules

- Schedule S is conflict serializable if S is conflict equivalent to SOME serial schedule!

Schedule 1

| T1:       | R(A), W(A), R(B), W(B) |
| T2:       | R(A), W(A), R(B)       |

Schedule 2 (serial)

| T1:       | R(A), W(A), R(B), W(B) |
| T2:       | R(A), W(A), R(B)       |
Example

- A schedule that is not conflict serializable.

<table>
<thead>
<tr>
<th>Schedule</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>T1:</td>
<td>R(A), W(A), R(B), W(B)</td>
</tr>
<tr>
<td>T2:</td>
<td>R(A), W(A), R(B), W(B)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Serial1</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>T1:</td>
<td>R(A), W(A), R(B), W(B)</td>
</tr>
<tr>
<td>T2:</td>
<td>R(A), W(A), R(B), W(B)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Serial2</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>T1:</td>
<td>R(A), W(A), R(B), W(B)</td>
</tr>
<tr>
<td>T2:</td>
<td>R(A), W(A), R(B), W(B)</td>
</tr>
</tbody>
</table>
How to check conflict serializability?

**Precedence graph**: (a.k.a a serializability graph)

One node per transaction;

edge from $T_i$ to $T_j$ if $T_i$ has a conflicting action with $T_j$ and $T_i$ precedes $T_j$. 
Example

- A schedule that is not conflict serializable:

<table>
<thead>
<tr>
<th>T1:</th>
<th>R(A), W(A), R(B), W(B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2:</td>
<td>R(A), W(A), R(B), W(B)</td>
</tr>
</tbody>
</table>

The cycle in the graph reveals the problem. The output of T1 depends on T2, and vice-versa.

**Theorem:** Schedule is conflict serializable if and only if its precedence graph is acyclic (Proof by contradiction)
Recoverable schedules: a transaction $T$ is not allowed to Commit until all other transactions that wrote values that $T$ read has committed.

Is a or b recoverable?

(a)

(b)
Recoverable schedule that illustrates a cascaded abort. $T_3$ aborts, forcing $T_2$ to abort, which then forces $T_1$ to abort. (Cascading Aborts)
Strict 2PL

- **Strict Two-phase Locking (Strict 2PL) Protocol:**
  - Each transaction must obtain a \(S\) (shared) lock on object before reading, and an \(X\) (exclusive) lock on object before writing.
  - All locks held by a transaction are released when the transaction completes.
  - If a transaction holds an \(X\) lock on an object, no other transaction can get a lock (\(S\) or \(X\)) on that object.

- **Strict 2PL allows only schedules whose precedence graph is acyclic.** (Proof by contradiction)
Two-Phase Locking (2PL) (non-strict)

- Two-Phase Locking Protocol
  - Each transaction must obtain a S (shared) lock on object before reading, and an X (exclusive) lock on object before writing.
  - A transaction can not request additional locks once it releases any locks.
  - If a transaction holds an X lock on an object, no other transaction can get a lock (S or X) on that object.
Strict vs non-strict 2PL

- Does strict and non-strict 2PL produce serializable schedules?
- Does strict 2PL avoid cascading aborts?
- Does strict 2PL produce only recoverable schedules?
- How about non-strict 2PL?
Lock Management

- Lock and unlock requests are handled by the lock manager
- Lock table entry:
  - Number of transactions currently holding a lock
  - Type of lock held (shared or exclusive)
  - Pointer to queue of lock requests
- Locking and unlocking have to be atomic operations
- Lock upgrade: transaction that holds a shared lock can be upgraded to hold an exclusive lock
Deadlocks

- **Deadlock**: Cycle of transactions waiting for locks to be released by each other.
- **Two ways of dealing with deadlocks:**
  - Deadlock prevention
  - Deadlock detection
Deadlock Detection

- Create a **wait-for graph**:
  - Nodes are transactions
  - There is an edge from Ti to Tj if Ti is waiting for Tj to release a lock
- Periodically check for cycles in the waits-for graph
Deadlock Detection (Continued)

Example:

T1: S(A), R(A), S(B)
T2: X(B), W(B), X(C)
T3: S(C), R(C)
T4: X(B)

T1  T2

T4  T3
Deadlock Detection (Continued)

Example:

T1:  S(A), R(A),  S(B)
T2:  X(B), W(B)    X(C)
T3:  S(C), R(C)
T4:  X(B)
Deadlock Detection (Continued)

Example:

T1:  S(A), R(A), S(B)
T2:  X(B), W(B)  X(C)
T3:  S(C), R(C)
T4:  X(B)
Deadlock Detection (Continued)

Example:

T1: S(A), R(A),
    S(B)
T2: X(B), W(B)
    X(C)
T3: S(C), R(C)
T4: X(B)
Deadlock Detection (Continued)

Example:

T1:  S(A), R(A),  S(B)
T2:  X(B), W(B)  X(C)
T3:  S(C), R(C)  X(A)
T4:  X(B)

DEADLOCK!
Deadlock Prevention

- Assign priorities based on timestamps. Assume Ti wants a lock that Tj holds. Two policies are possible:
  - **Wait-Die**: If Ti has higher priority, Ti waits for Tj; otherwise Ti aborts
  - **Wound-wait**: If Ti has higher priority, Tj aborts; otherwise Ti waits
- If a transaction re-starts, make sure it has its original timestamp
Multiple-Granularity Locks

- Hard to decide what granularity to lock (tuples vs. pages vs. tables).
- Shouldn’t have to decide!
- Data “containers” are nested:
Solution: New Lock Modes, Protocol

- Allow transactions to lock at each level, but with a special protocol using **intention locks**:

  - Before locking an item, transact must set “intention locks” on all its ancestors.
  - For unlock, go from specific to general (i.e., bottom-up).
  - **SIX mode**: Like S & IX at the same time.
Multiple Granularity Lock Protocol

- Each transaction starts from the root of the hierarchy.
  - To get S or IS lock on a node, must hold IS or IX on parent node.
    - What if Xact holds SIX on parent? S on parent?
  - To get X or IX or SIX on a node, must hold IX or SIX on parent node.
  - Must release locks in bottom-up order.

Protocol is correct in that it is equivalent to directly setting locks at the leaf levels of the hierarchy.
Examples

- T1 scans R, and updates a few tuples:
  - T1 gets an SIX lock on R, then repeatedly gets an S lock on tuples of R, and occasionally upgrades to X on the tuples.

- T2 uses an index to read only part of R:
  - T2 gets an IS lock on R, and repeatedly gets an S lock on tuples of R.

- T3 reads all of R:
  - T3 gets an S lock on R.
  - OR, T3 could behave like T2; can use lock escalation to decide which.

\[
\begin{array}{cccccc}
 & -- & IS & IX & S & X \\
-- & \checkmark & \checkmark & \checkmark & \checkmark & \checkmark \\
IS & \checkmark & \checkmark & \checkmark & \checkmark & \\
IX & \checkmark & \checkmark & \checkmark & \\
S & \checkmark & \checkmark & \checkmark & \\
X & \checkmark & & & \\
\end{array}
\]
Optimistic CC (Kung-Robinson)

- Locking is a conservative approach in which conflicts are prevented. Disadvantages:
  - Lock management overhead.
  - Deadlock detection/resolution.
  - Lock contention for heavily used objects.
- If conflicts are rare, we might be able to gain concurrency by not locking, and instead checking for conflicts before transactions commit.
Kung-Robinson Model

- Transactions have three phases:
  - **READ**: transaction read from the database, but make changes to private copies of objects.
  - **VALIDATE**: Check for conflicts.
  - **WRITE**: Make local copies of changes public.
Validation

- Test conditions that are **sufficient** to ensure that no conflict occurred.
- Each transaction is assigned a numeric id.
  - Just use a **timestamp**.
- Transaction ids assigned at end of READ phase, just before validation begins.
- $\text{ReadSet}(Ti)$: Set of objects read by transact $Ti$.
- $\text{WriteSet}(Ti)$: Set of objects modified by $Ti$. 
Test 1

- For all $i$ and $j$ such that $T_i < T_j$, check that $T_i$ completes before $T_j$ begins.
Test 2

For all $i$ and $j$ such that $Ti < Tj$, check that:
- $Ti$ completes before $Tj$ begins its Write phase
- $WriteSet(Ti) \cap ReadSet(Tj)$ is empty.

Does $Tj$ read dirty data? Does $Ti$ overwrite $Tj$’s writes?
Test 3

- For all i and j such that Ti < Tj, check that:
  - Ti completes Read phase before Tj does
  - WriteSet(Ti) $\cap$ ReadSet(Tj) is empty
  - WriteSet(Ti) $\cap$ WriteSet(Tj) is empty.

Does Tj read dirty data? Does Ti overwrite Tj’s writes?
Overheads in Optimistic CC

- Must record read/write activity in ReadSet and WriteSet per transaction.
  - Must create and destroy these sets as needed.
- Must check for conflicts during validation, and must make validated writes \``global\''.
  - Critical section can reduce concurrency.
  - Scheme for making writes global can reduce clustering of objects.
- Optimistic CC restarts transactions that fail validation.
  - Work done so far is wasted; requires clean-up.
Idea: Give each object a read-timestamp (RTS) and a write-timestamp (WTS), give each transaction a timestamp (TS) when it begins:

- If action $a_i$ of transaction $T_i$ conflicts with action $a_j$ of transaction $T_j$, and $TS(T_i) < TS(T_j)$, then $a_i$ must occur before $a_j$. Otherwise, restart violating transaction.
When transact T wants to read Object O

- If TS(T) < WTS(O), this violates timestamp order of T w.r.t. writer of O.
  - So, abort T and restart it with a new, larger TS. (If restarted with same TS, T will fail again! Contrast use of timestamps in 2PL for ddlk prevention.)

- If TS(T) > WTS(O):
  - Allow T to read O.
  - Reset RTS(O) to max(RTS(O), TS(T))

- Change to RTS(O) on reads must be written to disk! This and restarts represent overheads.
When transact T wants to Write Object O

- If TS(T) < RTS(O), this violates timestamp order of T w.r.t. writer of O; abort and restart T.
- If TS(T) < WTS(O), violates timestamp order of T w.r.t. writer of O.
  - Thomas Write Rule: We can safely ignore such outdated writes; need not restart T! (T’s write is effectively followed by another write, with no intervening reads.) Allows some serializable but non-conflict serializable schedules:
- Else, allow T to write O.

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A)</td>
<td>W(A) Commit</td>
</tr>
<tr>
<td>W(A) Commit</td>
<td>Commit</td>
</tr>
</tbody>
</table>
Unfortunately, unrecoverable schedules are allowed:

- Timestamp CC can be modified to allow only recoverable schedules:
  - Buffer all writes until writer commits (but update WTS(O) when the write is allowed.)
  - Block readers T (where TS(T) > WTS(O)) until writer of O commits.
  - Similar to writers holding X locks until commit, but still not quite 2PL.
The **PHANTOM** Problem in RDBMS concurrency control
Implementation of Transactions

- A transaction starts with the execution of a SQL-Data statement assuming that there is no current transaction.
- Transaction ends with
  - COMMIT
  - ROLLBACK
Implementation of Transactions

update cust_accounts set balance = balance - 1500
   where account_no = '70-490930.1';

commit;

update cust_accounts set balance = balance + 1500
   where account_no = '70-909249.1';

commit;
## Transaction Support in SQL-92

- Each transaction has an access mode, a diagnostics size, and an isolation level.

<table>
<thead>
<tr>
<th>Isolation Level</th>
<th>Dirty Read</th>
<th>Unrepeatable Read</th>
<th>Phantom Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Uncommitted</td>
<td>Maybe</td>
<td>Maybe</td>
<td>Maybe</td>
</tr>
<tr>
<td>Read Committed</td>
<td>No</td>
<td>Maybe</td>
<td>Maybe</td>
</tr>
<tr>
<td>Repeatable Reads</td>
<td>No</td>
<td>No</td>
<td>Maybe</td>
</tr>
<tr>
<td>Serializable</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
QUIZ Number 4

- Answer the following question ....