Principles of Database Systems

Course Number: CSGY-6083
Section Number: B

Module 5

Instructor : **Amit Patel**
MS, PMP®, OCP®
Email: asp13@nyu.edu
patelamitnyu@gmail.com

Database Transactions

Backup and Recovery System

Learning Objectives

Database Transaction

- Transaction concept
- ACID properties of database transactions
- Transaction states
- Transaction concurrency
- Transaction serializability
- Implementation of transaction isolation
- Transaction Definition in SQL
- Transaction control for achieving concurrency
- Locking mechanism
- Dead Lock and it’s prevention methods

Transaction Concept

- A **transaction** is a *unit* of work execution that accesses and possibly updates various data items. A transaction can have one or more instructions to perform in order.

Transaction is the logical collection (Unit) of instructions in particular order to perform a specific tasks in database.

- E.g. transaction to transfer $50 from account A to account B:
  1. read(*A*)
  2. *A* := *A* – 50
  3. write(*A*)
  4. read(*B*)
  5. *B* := *B* + 50
  6. write(*B)*
- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions

Required Properties of a Transaction

- **Consider a transaction to transfer $50 from account A to account B:**
  1. read(A)
  2. $A := A - 50$
  3. write(A)
  4. read(B)
  5. $B := B + 50$
  6. write(B)
- **Atomicity requirement (ALL or NONE) [Everything or Nothing]**
  - **If the transaction fails after step 3 and before step 6, money will be "lost" leading to an inconsistent database state**
    - **Failure could be due to software or hardware**
  - **The system should ensure that updates of a partially executed transaction are not reflected in the database**

- **Durability requirement — once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures.**

Required Properties of a Transaction (Cont.)

- **Consistency requirement** in above example:
  - The sum of A and B is unchanged by the execution of the transaction

- In general, consistency requirements include
    - Explicitly specified integrity constraints such as primary keys and foreign keys
    - Implicit integrity constraints (via transaction control)
        - e.g., sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand

- A transaction, when starting to execute, must see a consistent database.

- When the transaction completes successfully the database must be consistent
  - Erroneous transaction logic can lead to inconsistency

Required Properties of a Transaction (Cont.)

- **Isolation requirement — if between steps 3 and 6 (of the fund transfer transaction) , another transaction T2 is allowed to access the partially updated database, it will see an inconsistent database (the sum *A + B* will be less than it should be).**
  A=1000 B=2000

T1	T2
1.read(*A*) → 1000	
2.*A* := *A* – 50 → 950	
3.write(*A*) → 950	
	read(A), read(B), print(A+B) → 950, 2000, 2950
4.read(*B*) → 2000	
5.*B* := *B* + 50 → 2050	
6.write(*B*) → *2050*	

- **Isolation can be ensured trivially by running transactions serially**
  - **That is, one after the other.**
- **However, executing multiple transactions concurrently has significant benefits, as we will see later.**

ACID Properties Of Tx

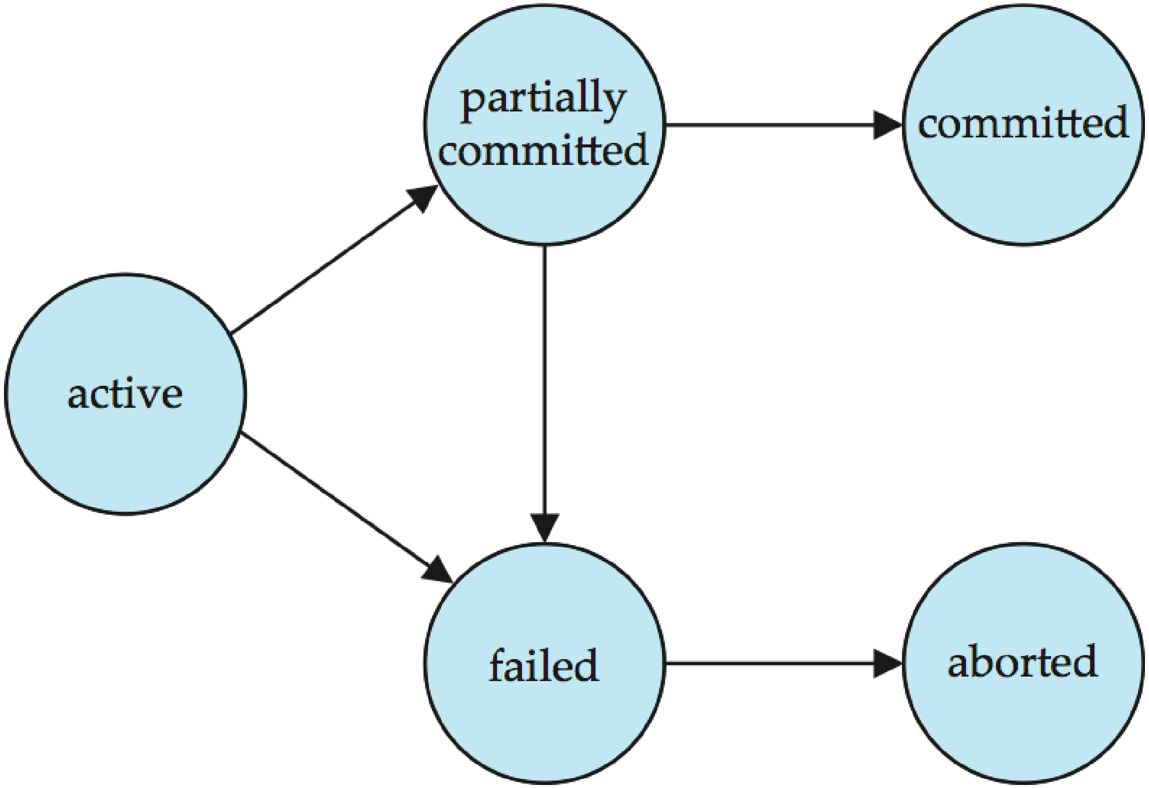
A transaction is a unit of work execution that accesses and possibly updates various data items. To preserve the integrity of data the database system must ensure:

- **Atomicity. Either all operations of the transaction are properly reflected in the database or none are.**
- **Consistency. Execution of a transaction in isolation preserves the consistency of the database.**
- **Isolation. Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.**
  - **That is, for every pair of transactions T_i and T_j, it appears to T_i that either T_j, finished execution before T_i started, or T_j started execution after T_i finished.**
- **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Transaction State

- **Active** – the initial state; the transaction stays in this state while it is executing
- **Partially committed** – after the final statement has been executed.
- **Failed** -- after the discovery that normal execution can no longer proceed.
- **Aborted** – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
  - Restart the transaction
    - can be done only if no internal logical error
  - Kill the transaction
- **Committed** – after successful completion.

Transaction State (Cont.)



Concurrent Executions

- **Multiple transactions are allowed to run concurrently in the system. Advantages are:**
  - **Increased processor and disk utilization, leading to better transaction *throughput***
    - **E.g. one transaction can be using the CPU while another is reading from or writing to the disk**
  - **Reduced average response time, for transactions: short transactions need not wait behind long ones.**
- **Concurrency control schemes – mechanisms to achieve isolation of Tx with database concurrency and optimize response time.**

We must allow concurrent transactions such a way they do not collide. This is achieved by a concurrency control scheme (database locks)

  - **That is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database**

Schedules

- **Schedule** – a sequences of transactions that specify the chronological order in which instructions of concurrent transactions are executed
  - A schedule for a set of transactions must consist of all instructions of those transactions
  - Must preserve the order in which the instructions appear in each individual transaction.
- A transaction that successfully completes its execution will have a commit/rollback instructions as the last statement
  - By default transaction assumed to execute commit instruction as its last step
- A transaction that fails to successfully complete its execution will have an abort(rollback) instruction as the last statement

Schedule 1

- **Let T_1 transfer \$50 from A to B, and T_2 transfer 10% of the balance from A to B.**
- **An example of a serial schedule in which T_1 is followed by T_2 :**

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

Schedule 2

- A **serial** schedule in which T_2 is followed by T_1 :

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

Schedule 3

- Let T_1 and T_2 be the transactions defined previously. The following schedule is not a serial schedule, but it is equivalent to Schedule 1.

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) commit	
	read (B) $B := B + temp$ write (B) commit

Note -- In schedules 1, 2 and 3, the sum "A + B" is preserved.

Schedule 4

- **The following concurrent schedule does not preserve the sum of "*A* + *B*"**

A = 1000 B = 500 A+B = 1500

	T_1	T_2	
A: 1000	read (A)		
A: 950	$A := A - 50$		
		read (A)	A: 1000
		$temp := A * 0.1$	temp: 100
		$A := A - temp$	A: 900
		write (A)	A: 900
		read (B)	B: 500
A: 900	write (A)		
B: 500	read (B)		
B=: 550	$B := B + 50$		
B: 550	write (B)		
A: 900 B: 550	commit		
		$B := B + temp$	B: 650
		write (B)	B: 650
		commit	A: 900 B: 650

A = 900 B = 650 A+B = 1550

Conflicting Instructions

- Let I_i and I_j be two Instructions of transactions T_i and T_j respectively. Instructions I_i and I_j 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.

T1 T2

1. I_i = read(Q), I_j = read(Q). I_i and I_j don't conflict.
2. I_i = read(Q), I_j = write(Q). They conflict.
3. I_i = write(Q), I_j = read(Q). They conflict
4. I_i = write(Q), I_j = write(Q). They conflict

Readers do not block other readers

READ/WRITE CONFLICT SCENARIOS: CONFLICTING DATABASE OPERATIONS MATRIX

	TRANSACTIONS		
	T1	T2	RESULT
Operations	Read	Read	No conflict
	Read	Write	Conflict
	Write	Read	Conflict
	Write	Write	Conflict

Concurrency Control

- **Serialization: A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency**

- **Concurrency-control schemes tradeoff between the amount of concurrency they allow and the amount of overhead that they incur (work involved by the mechanism to allow for concurrent transactions)**

- **Goal – to develop concurrency control protocols that will assure serialization, and thus ensure data consistency.**

Transaction Definition in SQL

- **Data manipulation language (Insert-Update-Delete) must include a construct for specifying the set of actions that comprise a transaction.**

- **In SQL, a transaction begins implicitly when:**
  - ✓ New session starts
  - ✓ At the end of previous transaction

  Transaction can start explicitly when you instruct BEGIN TRANSACTION and end explicitly when you instruct END TRANSACTION

- **A transaction in SQL ends by:**
  - ✓ **Commit:** commits current transaction, and begins a new transaction
  - ✓ **Rollback:** causes current transaction to abort, and begins a new transaction
  - ✓ **DDL/DCL commands:** Self commits and begins a next transaction
  - ✓ **Session close (exit/quit)** [if you exit/quit gracefully without committing TX, it will commit)

- **In almost all database systems, by default, every SQL statement also commits implicitly if it executes successfully (risk involved)**
  - **Implicit commit can be turned off by a database directive**
    - E.g. in JDBC, connection.setAutoCommit(false);
    - SET AUTO COMMIT ON/OFF (OFF is default setting in Oracle)

Practice Exercise

a) Connect to the database **TX1 started**

b) **SELECT empno, deptno, sal FROM ap_emp WHERE deptno=30;**

c) **UPDATE ap_emp SET sal=sal+300 WHERE deptno=30;**

d) **ALTER TABLE ap_emp ADD (birthdate date); TX1 completed, TX2 started**

e) **ROLLBACK; - TX2 completed, TX3 started**

f) **SELECT empno, deptno, sal FROM ap_emp WHERE deptno=30;**

g) **SELECT empno, deptno, sal FROM ap_emp WHERE deptno=20;**

h) **UPDATE ap_emp SET sal=sal+300 WHERE deptno=20;**

i) **COMMIT;**

j) **ROLLBACK;**

k) **SELECT empno, deptno, sal FROM ap_emp WHERE deptno=20;**

l) **SELECT empno, deptno, sal FROM ap_emp WHERE deptno=10;**

m) **UPDATE ap_emp SET sal=sal+100 WHERE deptno=10;**

n) **Disconnect from database (EXIT)**

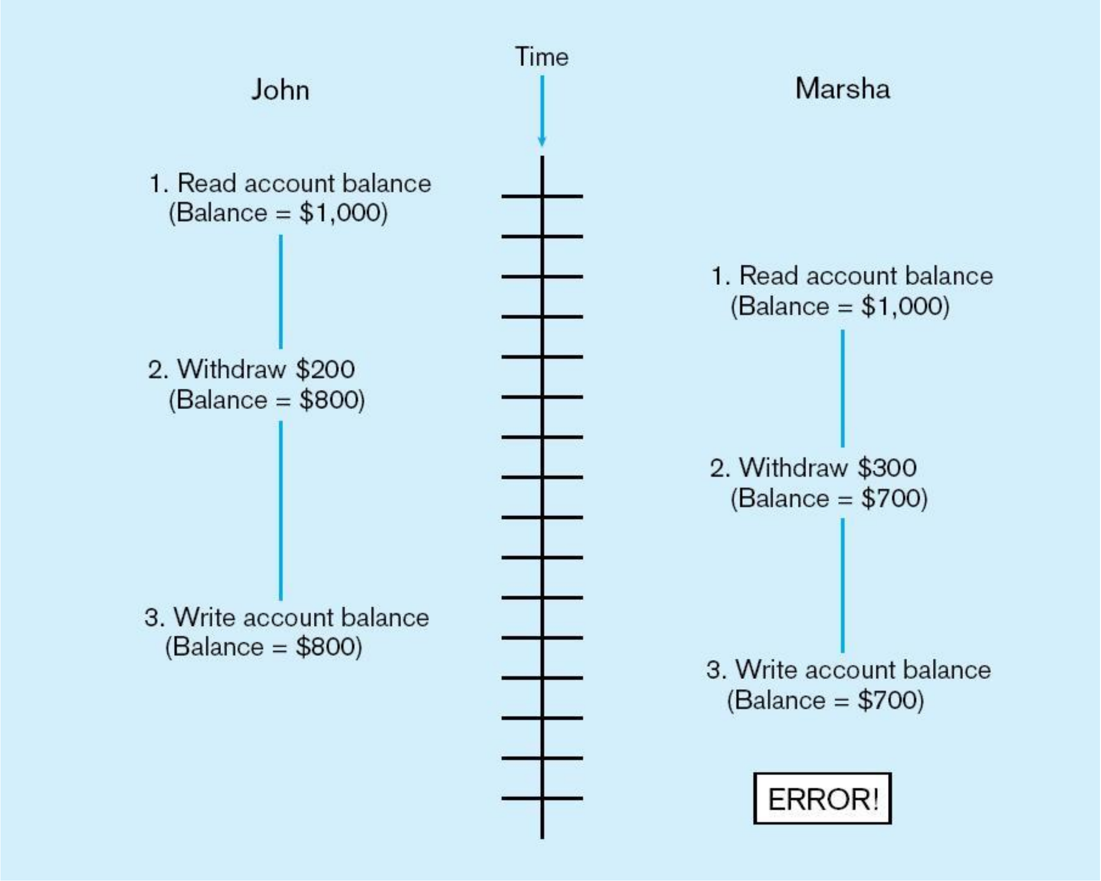
o) **Connect to the database**

p) **SELECT empno, deptno, sal FROM ap_emp WHERE deptno=10;**

For each new transaction give name to transaction, e.g. TX1, TX2, TX3.. , and state when each transaction started and when ended. What database changes will you see at as effect of all transactions?

Concurrency Control

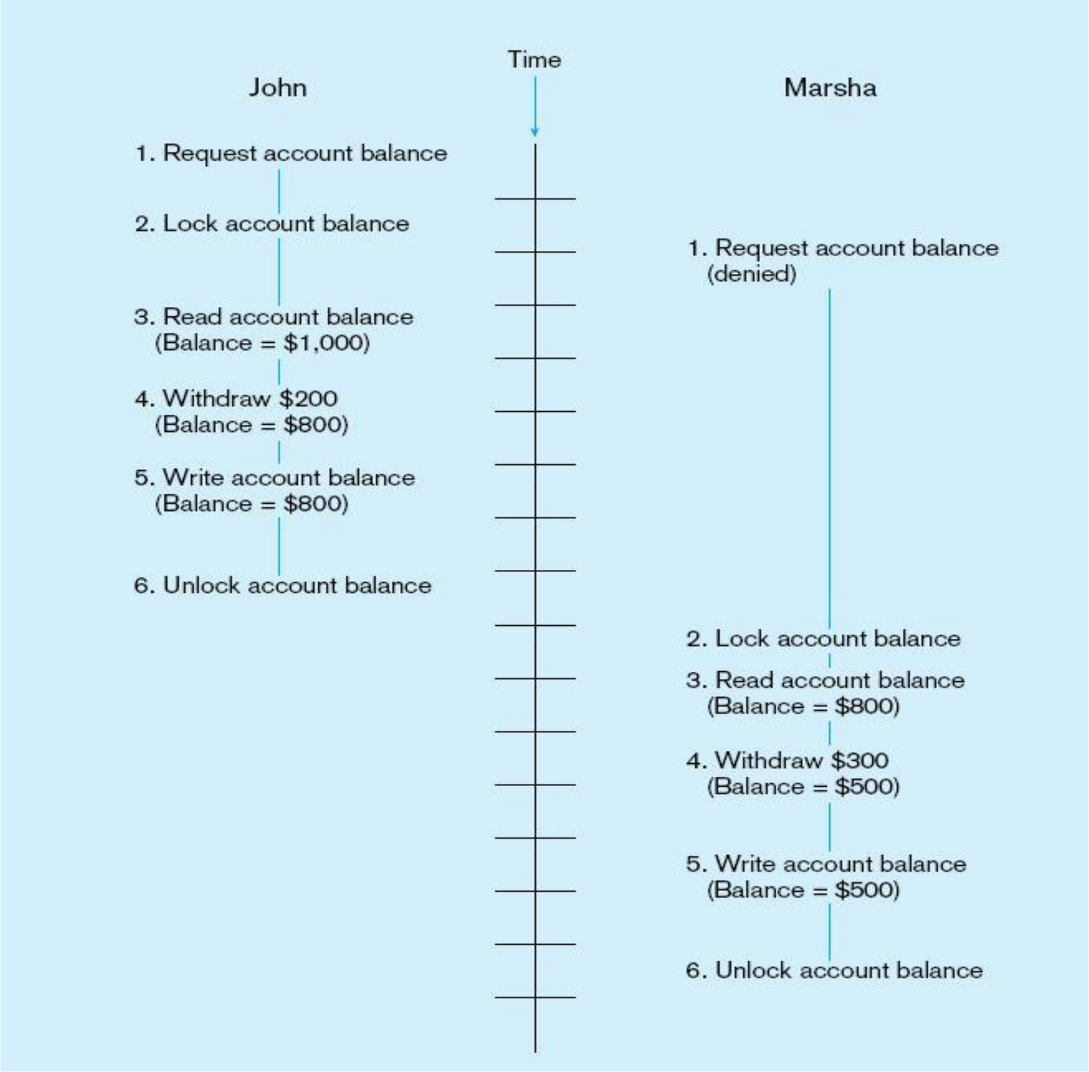
Lost update (no concurrency control in effect)



Concurrency Control Techniques

- **Serializability**
  - **Finish one transaction before starting another**
- **Locking Mechanisms**
  - **The most common way of achieving serialization**
  - **Data that is retrieved for the purpose of updating is locked for the update**
  - **No other user can perform update until unlocked**
  - **Unlocking happens on COMMIT/ROLLBACK**

Updates with locking (concurrency control)



This prevents the lost update problem

Locking Mechanisms

- **Locking level:**
  - Database–used during database updates
  - Table–used for bulk updates
  - Block or page–very commonly used
  - Record–only requested row; fairly commonly used
  - Field–requires significant overhead; impractical

Lock-Based Protocols

- **A lock is a mechanism to control concurrent access to a data item**

Lock Types: lock-S(Shared/Read) lock-X (Exclusive/Write)

- **Share lock (lock-S):**
  - Share locks can be placed on objects that do not have an exclusive lock already placed on them.
  - Prevents others from updating the data.
  - But still, others can read the data (others can place S-locks on it).
  - More than one share lock can be placed on the same object at the same time.
- **Exclusive lock (lock-X):**
  - Exclusive locks can only be placed on rows that do not have any other kind of lock (not even S-lock) on it.
  - Once an exclusive lock is placed on a row, no other locks (not even S-locks) can be placed on the same row anymore.
  - Prevents others from reading or updating the data.

Lock-Based Protocols (Cont.)

- **Lock-compatibility matrix**

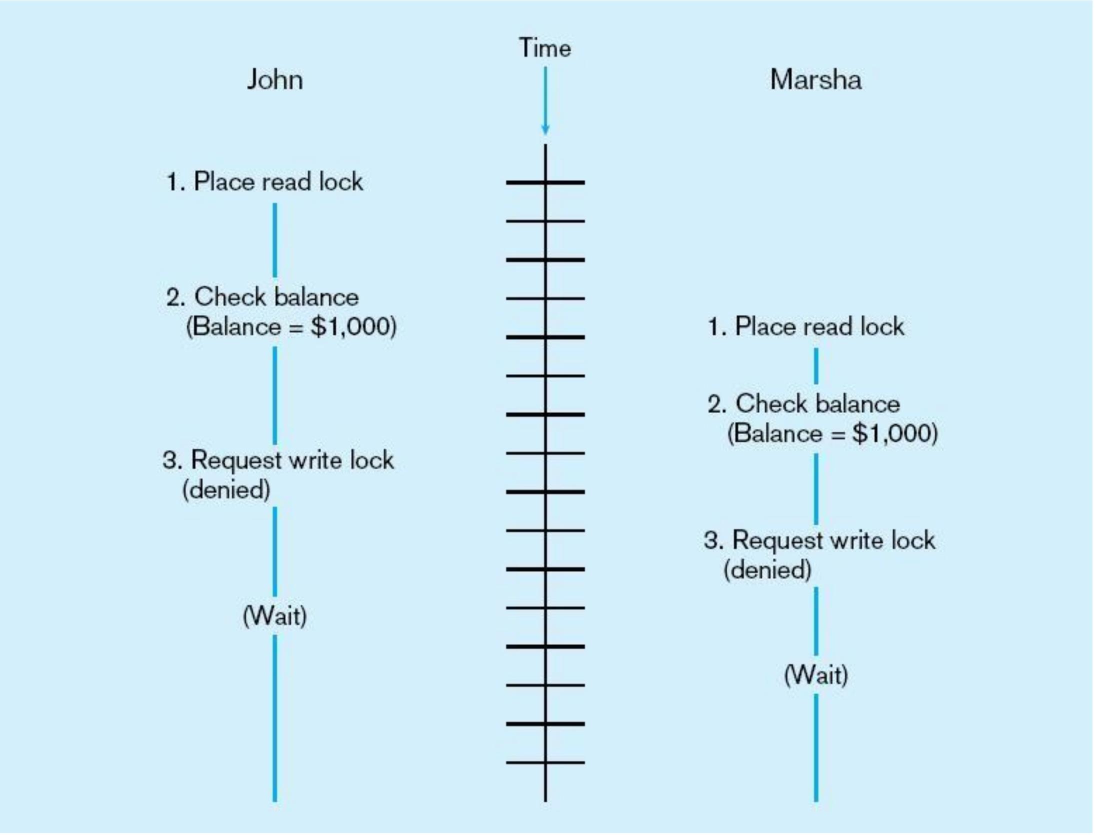
	S	X
S	true	false
X	false	false

- **A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions**
- **Any number of transactions can hold shared locks on an item,**
  - **But if any transaction holds an exclusive on the item no other transaction may hold any lock on the item.**
- **If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted.**

Deadlocks

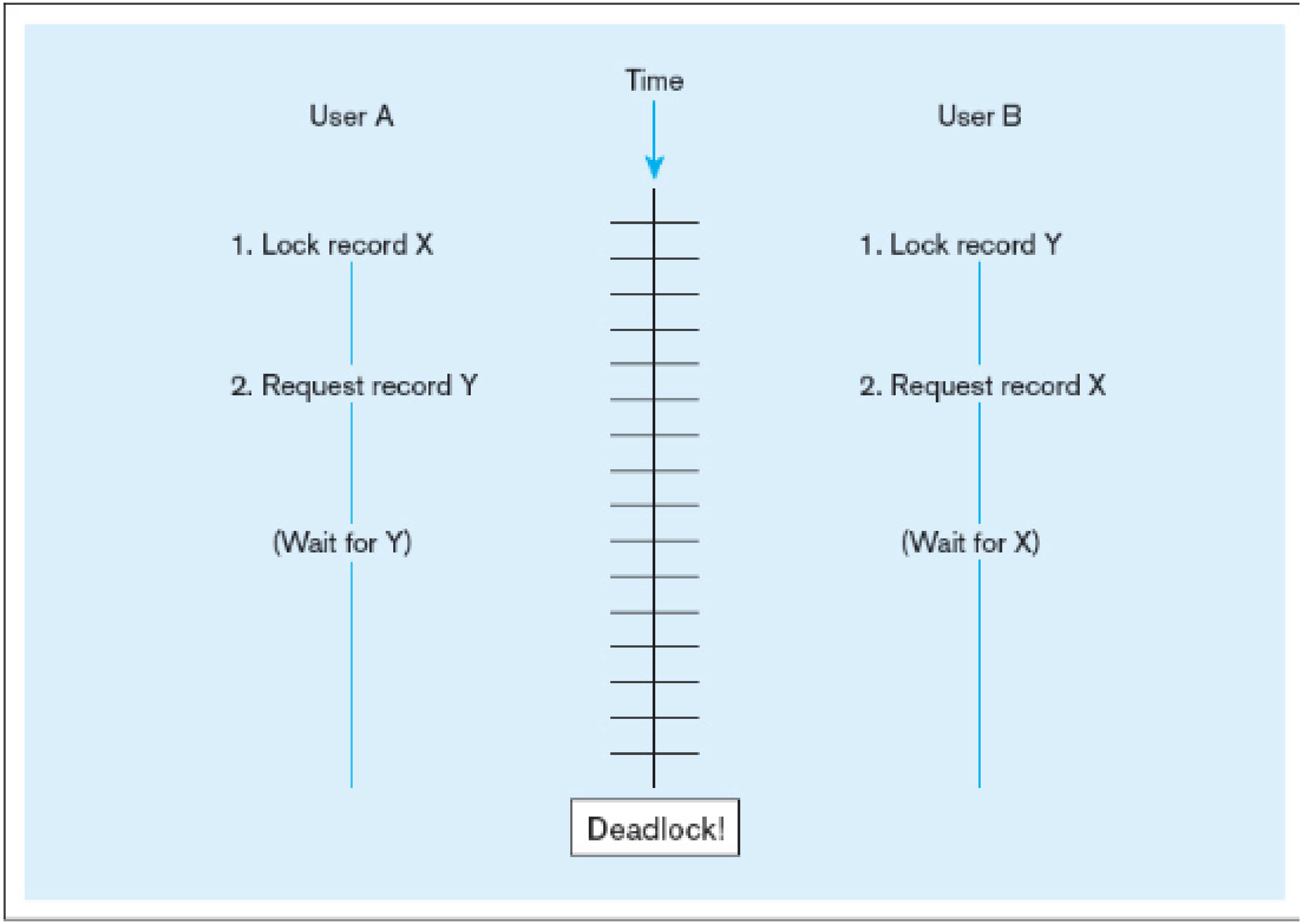
When two or more transactions have locked common resources, and each waits for the other to unlock their resources.

The problem of deadlock



John and Marsha will wait forever for each other to release their locked resources!

Deadlocks…


Time
User A
User B
1. Lock record X
2. Request record Y
(Wait for Y)
1. Lock record Y
2. Request record X
(Wait for X)
Deadlock!

Managing Deadlock

- **Deadlock prevention:**
  - Lock all records required at the beginning of a transaction (write application code in correct sequence of intended work)

```
SELECT * FROM EMP WHERE DEPTNO=20
FOR UPDATE; -- Lock x
UPDATE EMP SET SAL=SAL+100 WHERE DEPTNO=20;
COMMIT; -- Lock x
```

May be difficult to determine all needed resources in advance (not really)

- **Deadlock Resolution:**
  - Allow deadlocks to occur
  - Mechanisms for detecting and breaking them

ORA-00060 – dead lock error in Oracle

More Deadlock Prevention Strategies

- **Following schemes use transaction timestamps for the sake of deadlock prevention alone.**
- **wait-die scheme — non-preemptive (authority/power)**
  - **Younger transactions never wait for older ones (older means smaller timestamp); they are rolled back instead. Older transaction may wait for younger one to release data item. a transaction may die several times before acquiring needed data item**
- **wound-wait scheme — preemptive (defensive)**
  - **Younger transactions may wait for older ones. Older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it.**
  - **may be fewer rollbacks than *wait-die* scheme.**

Deadlock prevention (Cont.)

- Timeout-Based Schemes: (break deadlock once detected)
  - a transaction waits for a lock only for a specified amount of time. If the lock has not been granted within that time, the transaction is rolled back and need to be restarted.
  - Thus, deadlocks will not wait for longer and all transactions participating in dead-lock will be killed.

Deadlock prevention (Cont.)

By default, the server does not time out a transaction. That is, the server waits indefinitely for a transaction to complete. If you set a timeout value for transactions, if a transaction isn't completed within the configured time, the Application Server rolls back the transaction.

TRANSACTIONTIMEOUT *n units*

- *Units*

One of the following: S, SEC, SECS, SECOND, SECONDS, MIN, MINS, MINUTE, MINUTES, HOUR, HOURS, DAY, DAYS.

Example

- TRANSACTIONTIMEOUT 5 S

Deadlock prevention (Cont.)

Oracle provides the FOR UPDATE clause in SQL syntax to allow the developer to lock a set of Oracle rows for the duration of a transaction. The FOR UPDATE clause is generally used in cases where an online system needs to display a set of row data on a screen and they need to ensure that the data does not change before the end-user has an opportunity to update the data.

Essentially, the options were either "wait forever" or "don't wait. Oracle has added additional flexibility to the syntax by allowing the SQL to wait for a pre-defined amount of time for locked rows before aborting.

In this example we select a student row and wait up to 15 seconds for another session to release their lock:

select last_name, first_name

from

student

where student_id = 12345

FOR UPDATE WAIT 15;

Database Backup and Recovery System

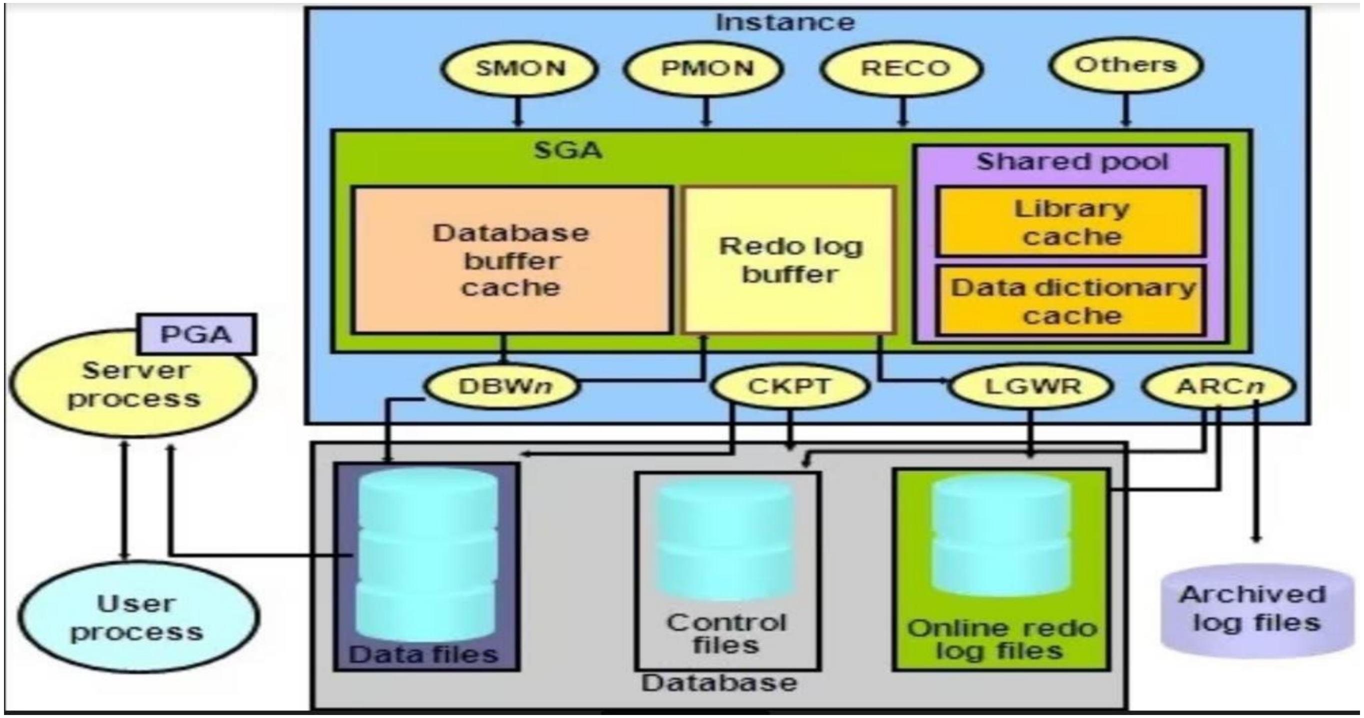
Objectives

- Oracle database server architecture
- Database recovery mechanism
    - Redo Logs,
    - Log switch,
    - Checkpoint,
    - Archive logs
- RTO and RPO for business continuity
- Backup strategies
- Backup methods
- Failure classification
- Log bases recovery
- Role of checkpoints in recovery

Why is it important to backup data?

What is important for data backups?

Oracle Database Server Architecture



At every COMMIT or every 3 seconds content of REDO LOG BUFFER flush to online REDO LOG FILE

Once Redo Log file becomes 2/3 full ,will create copy of it, called archived log, with sequentially generated number with timestamp,

ARCH_HRP_05012021_4554.log
ARCH_HRP_05012021_4555.log

Database Instance: Combination of allocated memory to database and mandatory background processes [volatile memory]   MUST BE UP AND RUNNING TO MAKE CONNECTIONS TO THE DATABASE

Database: Physical database [nonvolatile storage, Magnetic Storage]

Data files (actual changed data) + Control files + Redo log files (stores changes made to database – Only SQL Statements except SELECT) :   Redo log files are very important for data recovery.

AMM: Automatic memory management

SGA: Shared Global Area (System Global Area) = e.g., 6gb

Components: Shared Pool   => Library Cache, Dictionary Cache => 1gb =>
all statements, execution plans, dictionary query results

DB Buffer Cache => 4.8 GB =>> Data blocks copied to memory

Redo log buffer => 0.2 GB   [only SQL statements, except SELECT]

BACKGROUND PROCESSEES

========================

SMON: SYSTEM MONIOR

PMON: PROCESS MONITOR

RECO: RECOVERY

DBWR: DABASE WRITER (IT CAN BE MORE THAN ONE, DBWn [DBW1, DBW2....])

CKPT: CHEKPOINT

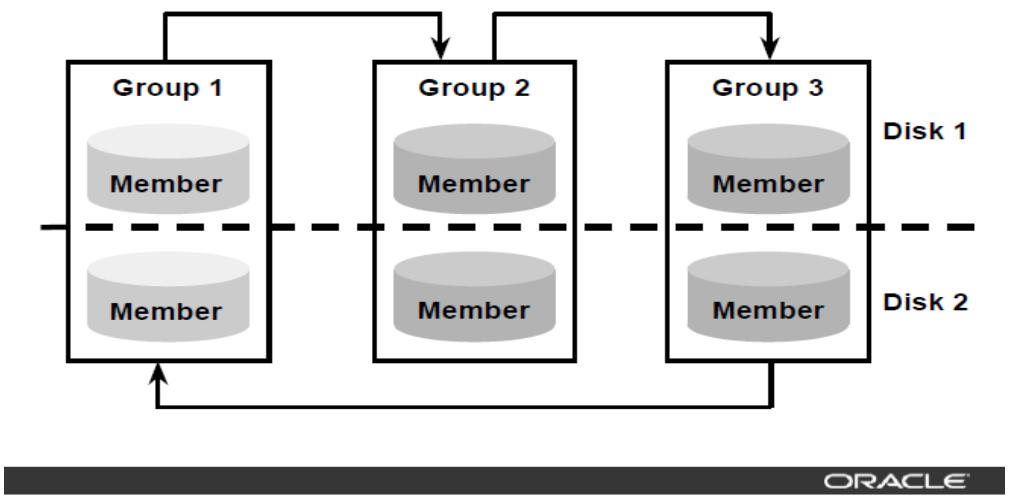
LGWR: LOG WRITER

ARCH: ARCHIVE (IT CAN BE MORE THAN ONE, ARCn [ARC1, ARC2....])

Redo Logs and Checkpoints

Structure of Redo Log Files

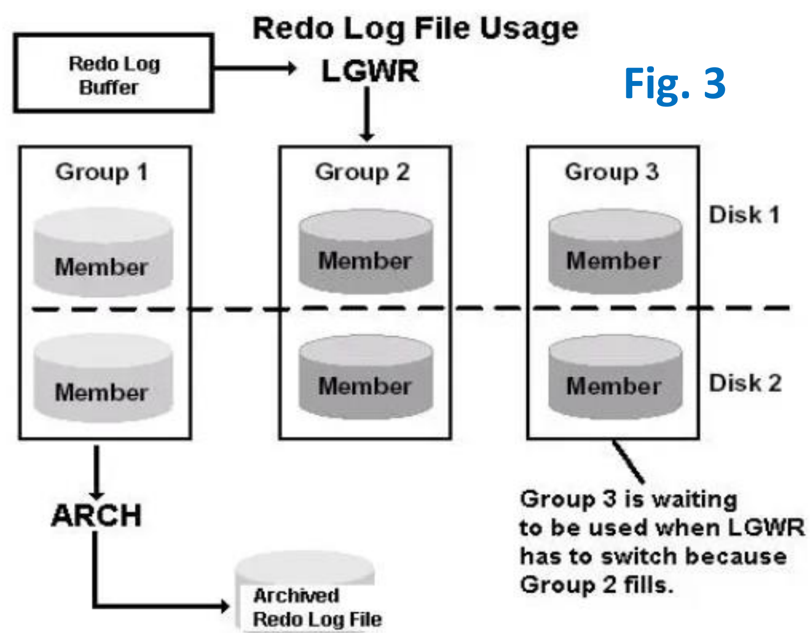
Fig. 1



Standard Redo Log Structure: Multiplexed (preferably on separate physical disks), and Three Log groups (minimum two)

Fig. 2

Fig. 3



REDO LOGS, LOG SWITCH, CHEKPOINT

Redo log buffer cache flush into Redo Logs files at every COMMIT / 3 seconds

Redo logs are written in cyclic fashion, and each log member in a particular log group is written simultaneously.

When Redo Log file becomes 2/3 full, the background process LGWR(Log Writer) switch writing to next Log Group. This is called LOG SWITCH.

At every LOG SWITCH

- **Log Sequence Number advances**
- **The current Redo Log Member Files become inactive, and next Redo Log Member files become active**
- **The ARCH background process, writes inactive redo log file to a offline archive log file**
- **Checkpoint occurs**

  At every checkpoint

  - **DBWR background process writes dirty buffers (modified buffers to physical data files on disks**
  - **CKPT background process writes SCN (System Checkpoint Number) to headers of Data Files and Control File.**

Back-up Strategies

Goal: restoring/recovering a database quickly and accurately after loss or damage

Restore and recovery are two separate steps.

- **Restore is the process of copying back data-files from the backup files.**

- **Recovery is the process of applying transaction information to the data-files to recover them to the state they were in just before the failure occurred.**

- **Strategy Considerations (RTO, RPO)**

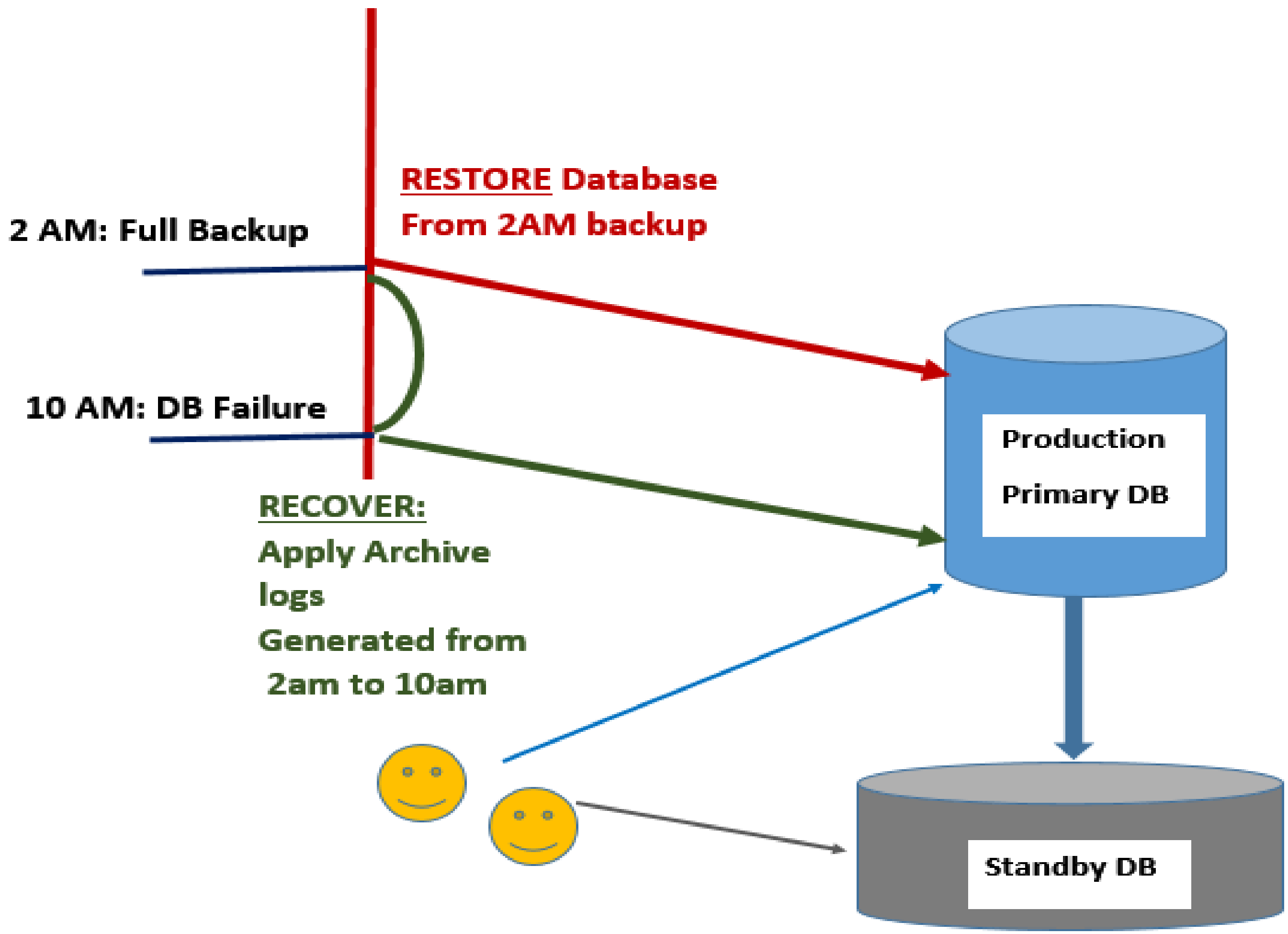
  RPO : Recovery Point Objectives

  RTO: Recovery Time Objectives

Restore and Recover scenario

RESTORE: Point in Time (DB pushed back to some time in past)

RECOVER: Point of Failure: DB will be roll forward from RESTORE point to the time it is failed.



Back-up Methods

- **Logical Backups: Metadata and/or Data export**
  [Only Restore, No Recovery] [at DB, Schema, Tables, Records level]
  [Not good for very large dataset] [very common use for data movements and data fixes]

- **Physical Backups:**
  - DBMS copy utility that produces backup copy of the entire database or subset
  - Cold backup (Full Offline Backup)–database is shut down during backup
    [Only Restore, No Recovery]
  - Hot backup(Full Online Backup)–Database is up during backup
  - Incremental backup–Database is up during backup, backup of changes since last backup (level0, level 1, cumulative)

  For HOT and Incremental backups DB must be in ARCHIVELOG MODE

- **Disaster Recovery** - Backups stored in secure, off-site location
  [Data protection against Natural or Intentional disaster]

Failure Classification

- **Transaction failure** :
  - **Logical errors: transaction cannot complete due to some internal error condition (e.g., constraint violations)**
  - **System errors: the database system must terminate an active transaction due to an error condition (e.g., deadlock)**

- **System crash: a power failure or other hardware or software failure causes the system to crash.**
  - **Fail-stop assumption: non-volatile storage contents are assumed to not be corrupted by system crash**

- **Disk failure: a head crash or similar disk failure destroys all or part of disk storage**
  - **Destruction is assumed to be detectable: disk drives use checksums to detect failures**

Log-Based Recovery

- A log is kept on stable storage.
  - The log is a sequence of log records and maintains a record of update activities on the database.
- When transaction T_i starts, it registers itself by writing a
  <T_i start>log record
- *Before* T_i executes write(X), a log record
  $<T_i, X, V_1, V_2>$
  is written, where V_1 is the value of X before the write (the old value), and V_2 is the value to be written to X (the new value).
- When T_i finishes it last statement successfully and committed, the log record <T_i *Commit*> is written.
- When T_i fails before its completion, the log record <T_i Abort> is written.
- Two approaches using logs
  - Deferred database modification
  - Immediate database modification (not preferable)

Database Modification

- The **immediate-modification** **scheme allows updates of an uncommitted transaction to be made to the buffer, or the disk itself, before the transaction commits**

- The **deferred-modification** **scheme performs updates to buffer/disk only at the time of transaction commit**
  - **Simplifies some aspects of recovery**
  - **But has overhead of storing local copy**

Undo and Redo Operations

- Undo of a log record <T_i, X, V_1, V_2> writes the old value V_1 to X
- Redo of a log record <T_i, X, V_1, V_2> writes the new value V_2 to X
- Undo and Redo of Transactions
  - **undo**(T_i) restores the value of all data items updated by T_i to their old values, going backwards from the last log record for T_i
  - **redo**(T_i) sets the value of all data items updated by T_i to the new values, going forward from the first log record for T_i

Immediate DB Modification Recovery Example

Below we show the log as it appears at three instances of time.

<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 1000, 950>	<T_0, A, 1000, 950>	<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 700, 600>	<T_1, C, 700, 600>
		<T_1 commit>
(a)	(b)	(c)

Recovery actions in each case above are:

(a) undo (T_0): B is restored to 2000 and A to 1000, and log records <T_0, B, 2000>, <T_0, A, 1000>, <T_0, abort> are written out

(b) redo (T_0) and undo (T_1): *A* and *B* are set to 950 and 2050 and C is restored to 700. Log records <T_1, C, 700>, <T_1, abort> are written out.

(c) redo (T_0) and redo (T_1): A and B are set to 950 and 2050 respectively. Then *C* is set to 600

Checkpoints

- **Streamline recovery procedure by periodically performing checkpointing**

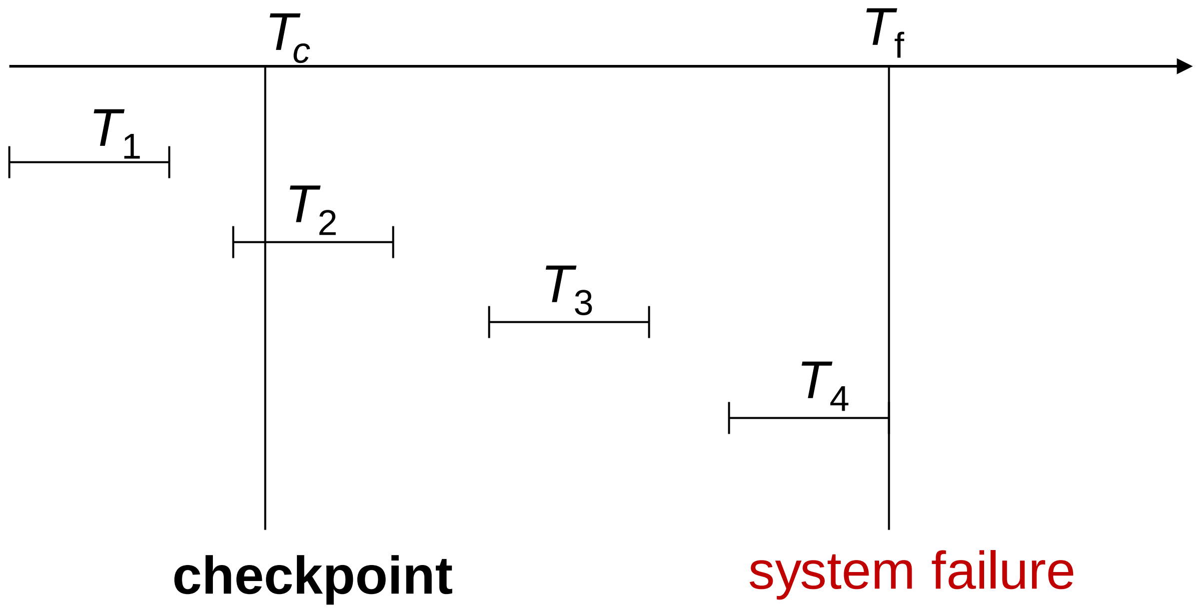
1. **Output all log records currently residing in main memory onto stable storage.**
2. **Output all modified buffer blocks to the disk.**
3. **Write a log record < checkpoint *L*> onto stable storage where *L* is a list of all transactions active at the time of checkpoint.**
   - **All updates are stopped while doing check pointing**

You can manually make log switch and checkpoint using following.

ALTER SYSTEM SWITCH LOGFILE;
ALTER SYSTEM CHECKPOINT;

(When should we do manual log switch and checkpoint?)

Example of Checkpoints



- T_1 can be ignored (updates already output to disk due to checkpoint)
- T_2 and T_3 redone.
- T_4 undone

Scan backwards from end of log to find the most recent <checkpoint *L*> record, only transactions that are in *L* or started after the checkpoint need to be redone or undone

This work is protected by United States copyright laws and is provided solely for the use of instructors in teaching their courses and assessing student learning. Dissemination or sale of any part of this work (including on the World Wide Web) will destroy the integrity of the work and is not permitted. The work and materials from it should never be made available to students except by instructors using the accompanying text in their classes. All recipients of this work are expected to abide by these restrictions and to honor the intended pedagogical purposes and the needs of other instructors who rely on these materials.