TRANSACTION MANAGEMENT

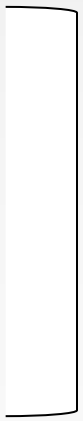
Chapters 15-17: Transaction Management

- Transaction (Chapter 15)
  - Transaction Concept
  - Transaction State
  - Concurrent Executions
  - Serializability
  - Recoverability
  - Testing for Serializability
- Concurrency control (Chapter 16)
  - Lock-based protocols
  - Timestamp-based protocols
  - Multiple granularity
  - Multiversion schemes
- Recovery Systems (Chapter 17)
  - Log-based recovery
  - Recovery with concurrent transactions
- Transaction in SQL
- Transaction management in Oracle 10g

Transaction Concept

- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.
- A transaction is a program including a collection of database operations, executed as a logical unit of data processing.
- Each high level operation can be divided into a number of low level tasks or operations. For example, a data update operation can be divided into three tasks −
- read_item() − reads data item from storage to main memory.
- modify_item() − change value of item in the main memory.
- write_item() − write the modified value from main memory to storage
- E.g. transaction to transfer €50 from account A to account B:

1. **read_from_account**(*A*)
2. *A* := *A* – 50
3. **write_to_account**(*A*)
4. **read_from_account**(*B*)
5. *B* := *B* + 50
6. **write_to_account**(*B*)



Transaction

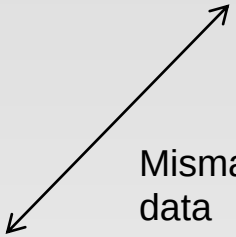
- Two main issues to deal with:
  Failures of various kinds, such as hardware failures and system crashes
  Concurrent execution of multiple transactions

Transaction Operations

- The low level operations performed in a transaction are −
- **begin_transaction** − A marker that specifies start of transaction execution.
- **read_item or write_item** − Database operations that may be interleaved with main memory operations as a part of transaction.
- **end_transaction** − A marker that specifies end of transaction.
- **commit** − A signal to specify that the transaction has been successfully completed in its entirety and will not be undone.
- **rollback** − A signal to specify that the transaction has been unsuccessful and so all temporary changes in the database are undone. A committed transaction cannot be rolled back.

Transaction ACID properties

- E.g. transaction to transfer €50 from account A to account B:
  1. **read_from_acoount**(*A*) (Suppose A=100,B=200 A+B=**300**)
  2. *A* := *A* – 50 (A=50)
  3. **write_to_account**(*A*) <---(Power Failure)
  4. **read_from_accont**(*B*)
  5. *B* := *B* + 50
  6. **write_to_account**(*B*) *(A+B=50+200=**250**)*

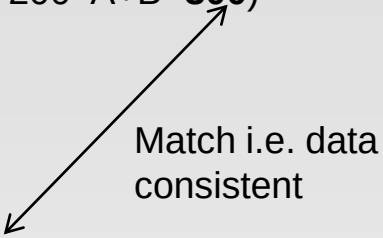


Mismatch i.e. data inconsistent

- **Atomicity requirement**
  - 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
  - **All or nothing**, regarding the execution of the transaction
- **Durability requirement** — once the user has been notified of transaction has completion, the updates must persist in the database even if there are software or hardware failures.

Transaction ACID properties (Cont.)

- Transaction to transfer €50 from account A to account B:
  1. **read_from_acoount**(*A*) (Suppose A=100,B=200 A+B=**300**)
  2. *A* := *A* – 50 (A=100-50=50)
  3. **write_to_account**(*A*)
  4. **read_from_accont**(*B*)
  5. *B* := *B* + 50 (B=200+50=250)
  6. **write_to_account**(*B*) *(A+B=50+250=**300**)*



- **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
    - e.g. sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand
  - A transaction must see a consistent database and must leave a consistent database
  - During transaction execution the database may be temporarily inconsistent.
    - Constraints to be verified only at the end of the transaction

Transaction ACID properties (Cont.)

- **Isolation requirement** — if between steps 3 and 6, 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).

	T1	T2	
1.	**read**(*A*)		(A=100)
2.	*A* := *A* – 50		(A=50)
3.	**write**(*A*)		
		read(A), read(B), print(A+B)	
	(A+B=50+200=250)		
4.	**read**(*B*)		
5.	*B* := *B* + 50		
6.	**write**(*B*)		

- 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 - Summary

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

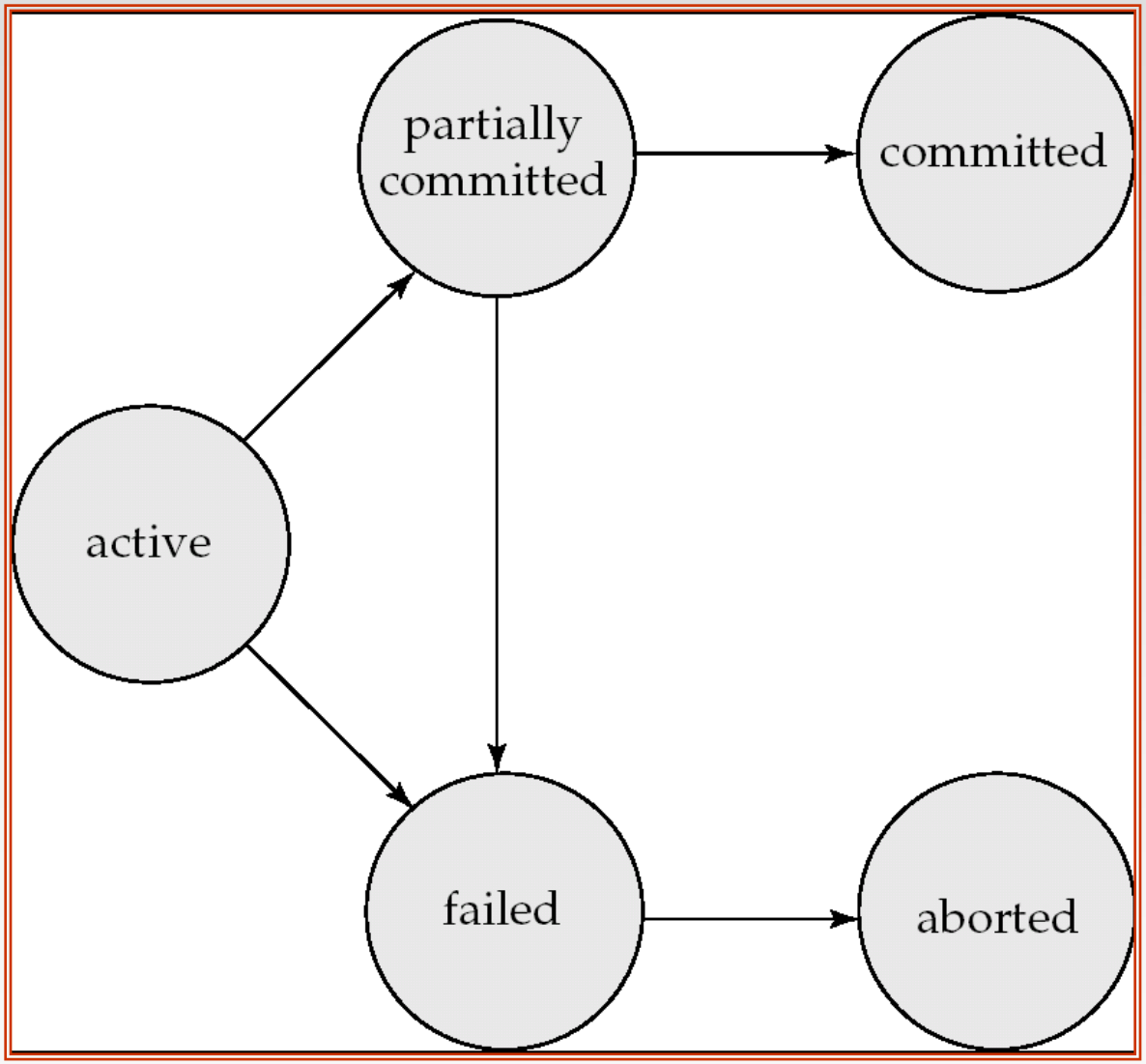
- **Atomicity** Either all operations of the transaction are properly reflected in the database or none are.
- **Consistency** Execution of a (single) transaction preserves the consistency of the database.
- **Isolation** Although multiple transactions may execute concurrently, each transaction must 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.

- To guarantee atomicity, external observable action should all be performed (in order) after the transaction is committed.

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
  - that is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database
    - Two-phase lock protocol
    - Timestamp-Based Protocols
    - Validation-Based Protocols
  - Studied in Operating Systems, and briefly summarized later

Schedules

- **Schedule** – a sequences of instructions 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 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 instruction as the last statement

- The goal is to find schedules that preserve the consistency.

Example Schedule 1

- Let T_1 transfer €50 from A to B, and T_2 transfer 10% of the balance from A to B.
- 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)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

A=100
A=50

B=200
B=250
SUM=300
A= 50
temp=50*.10=5
A=45
B=250
B=250+5=255

A+B=300

Equivalent Schedule

$T1$	$T2$
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)

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)

Example Schedule 2

- A serial schedule where 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)
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	

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

SupposeA=100,B=200
A=100
A=50

temp=50*0.1=5
A=50-5=45

B=200
B=200+50=250

B=250
B=250+5=255
A+B=45+255=300

In Schedules 1, 2 and 3, the sum A + B is preserved.

Example Schedule 4

- The following concurrent schedule does not preserve the value of ($A + B$).

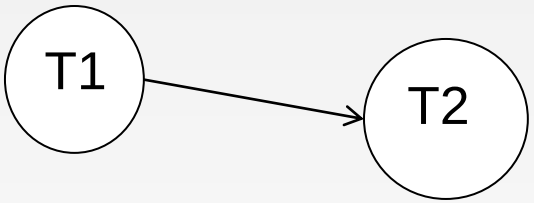
T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
	$B := B + temp$
	write(B)

SupposeA=100,B=200
A=100
A=50
A=100
temp=100*.10=10
A=100-10=90
B=200

A=**90**
B=200
B=200+50
B=250
B=250+10=**260**

Drawing Precedence Graph-Example

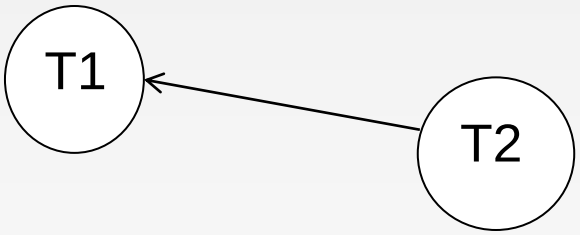
T1	T2
R(A)	
W(A)	
	R(A)
	W(A)



Precedence Graph

Drawing Precedence Graph-Example

T1	T2
	R(A)
	W(A)
R(A)	
W(A)	



Drawing Precedence Graph-Example

T1	T2
R(A)	
	R(A)
	W(A)
W(A)	

T1--->T2
OR
T2--->T1

Serializability Example

T1	T2	T3
	R(A)	
		R(A)
		W(A)
	W(A)	
R(A)		
W(A)		
	W(A)	

T1--->T2--->T3
T1---->T3---->T2
T2---->T1----->T3
T2----->T3---->T1
T3----->T1----->T2
T3----->T2------>T1

Factorial of no. of tractions
3!=6

Conflict View

Numerical

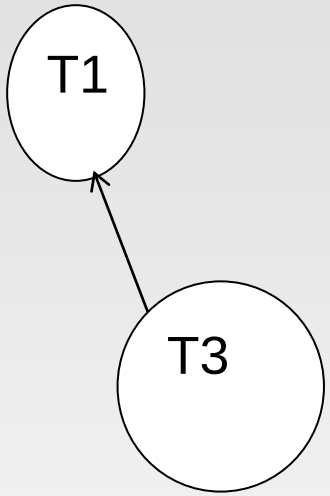
T1	T2	T3
R(X)		
		R(Y)
		R(X)
	R(Y)	
	R(Z)	
		W(Y)
	W(Z)	
R(Z)		
W(X)		
W(Z)		

R(X)--->W(X)
W(X)---->W(X)
W(X)-----> R(X)

Check conflicting pairs in other transaction

Calculating Indegree

- Example



Indegree=0

T2--->T3--->T1

Conflict Equivalent Schedules

S

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	

S’

T1	T2
R(A)	
W(A)	
R(B)	
	R(A)
	W(A)

R(A) --> R(A) Non-Conflicting Pair

R(A) --> W(A)
W(A) --> W(A)
W(A) --> R(A)
} Conflicting Pair

R(B) --> R(A)
W(B) --> R(A)
R(B) --> W(A)
W(A) ---> W(B)
} Non-Conflicting Pair

If S≡S’
then S is Conflict Serializable

Conflict Equivalent Schedules

S

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	

S

T1	T2
R(A)	
W(A)	
	R(A)
R(B)	
	W(A)

Swap these two operations in Table1

Swapped-Table2

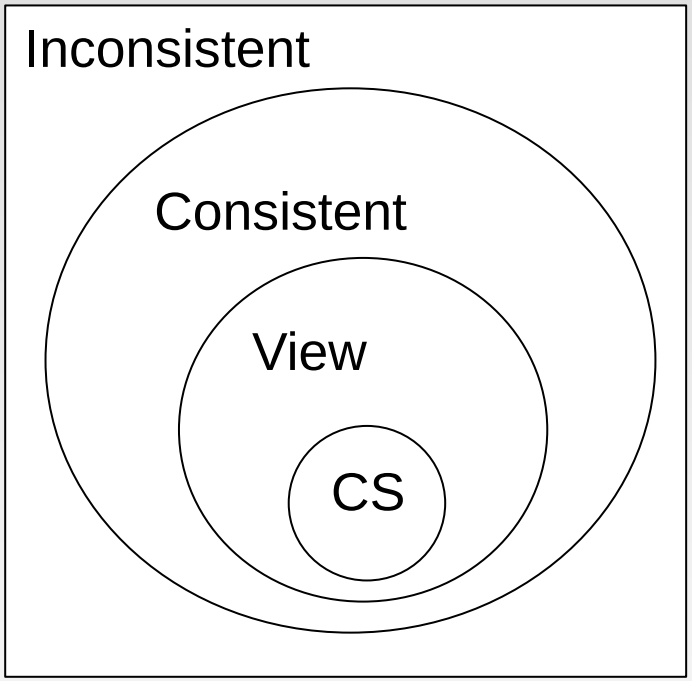
T1	T2
R(A)	
W(A)	
	R(A)
R(B)	
	W(A)

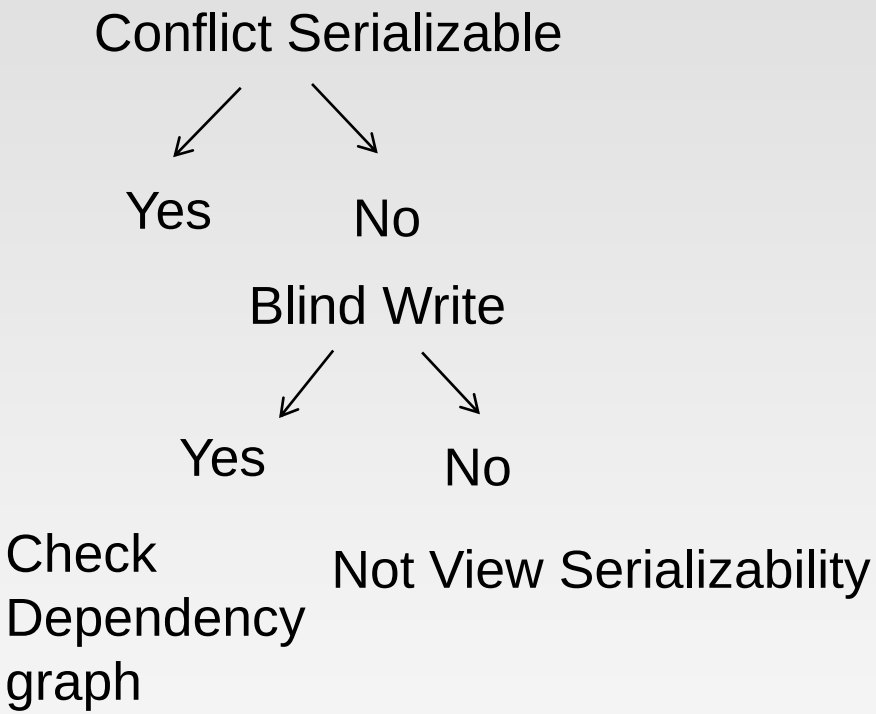
T1	T2
R(A)	
W(A)	
R(B)	
	R(A)
	W(A)

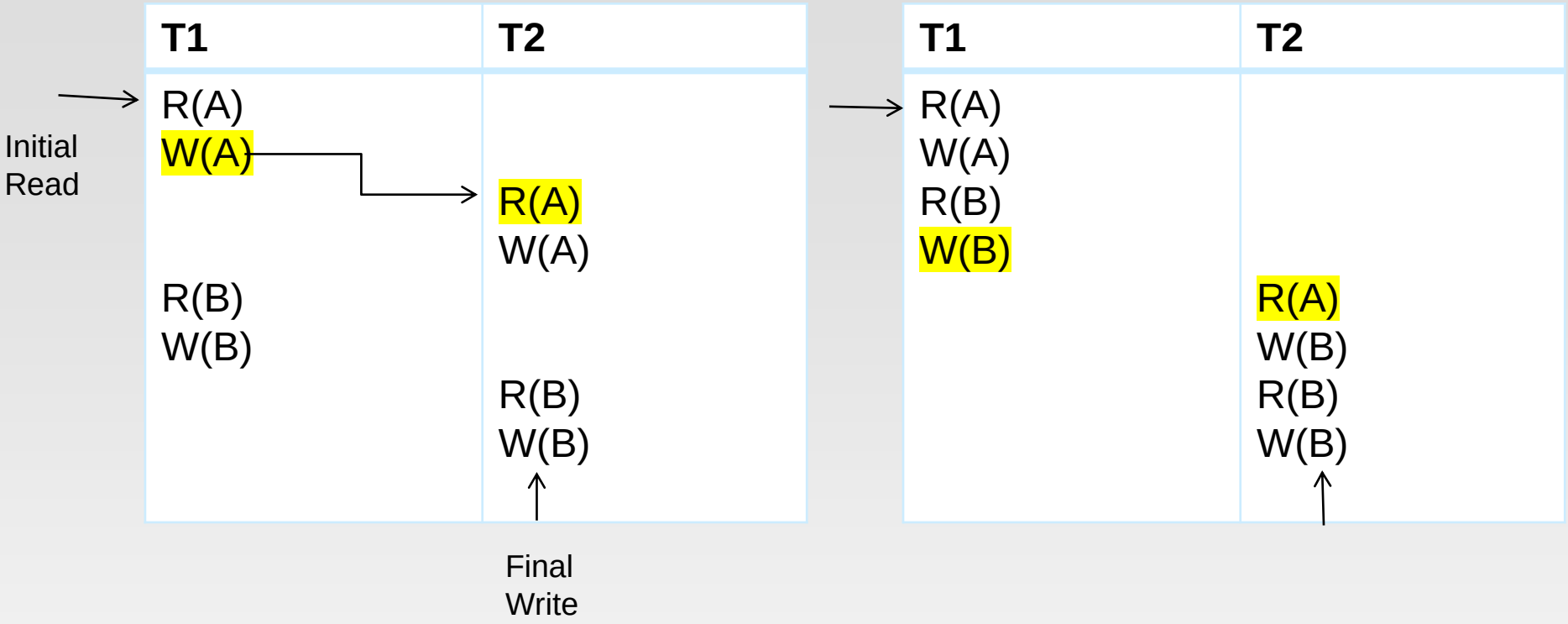
Swap these two operations in Table 2

Final Result equivalent to S'

View Serializability







For S--->VS, we need to check whether it is equivalent with any posssible serial execution schedule

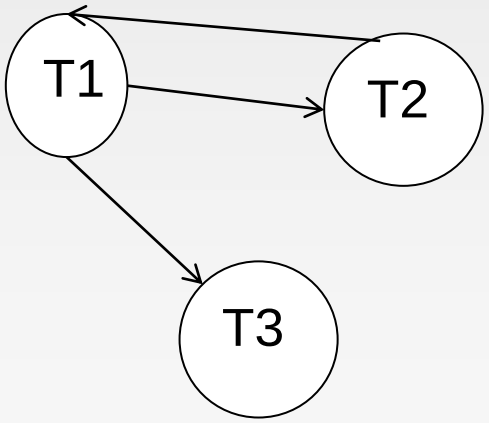
Properties of View Equivalence

- Initial Read
- Final Write
- Update Read

S

T1	T2	T3
R(A) W(A)	W(A)	 W(A)

T1	T2	T3
R(A) W(A)	 W(A)	 W(A)



1-2-3
1-3-2
2-1-3
2-3-1
3-1-2
3-2-1

Serializability

- **Goal :** Deal with concurrent schedules that are equivalent to some serial execution:
  - **Basic Assumption** – Each transaction preserves database consistency.
  - Thus serial execution of a set of transactions preserves database consistency.
- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:
  1. **conflict serializability**
  2. **view serializability**
- *Simplified view of transactions*
  - We ignore operations other than **read** and **write** instructions
  - We assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes.
  - Our simplified schedules consist of only **read** and **write** instructions.

Conflicting Instructions

- 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.
  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
- Intuitively, a conflict between I_i and I_j forces an order between them.
  - If I_i and I_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

Conflict Serializability

- If a schedule S can be transformed into a schedule S' by a series of swaps of non-conflicting instructions, we say that S and S' are **conflict equivalent**.
- We say that a schedule S is **conflict serializable** if it is conflict equivalent to a serial schedule
- Schedule 3 can be transformed into Schedule 6, a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions. Therefore it is conflict serializable.

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Schedule 3

T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Schedule 6

Conflict Serializability (Cont.)

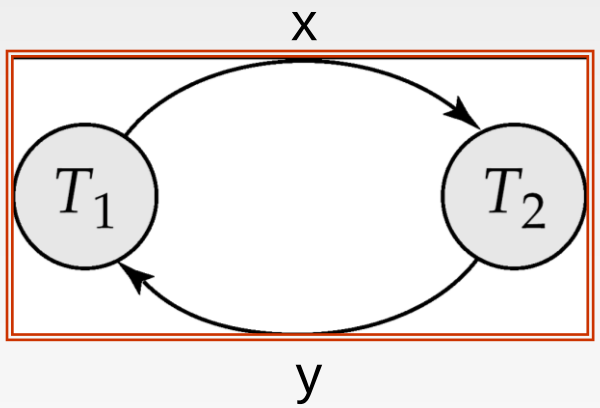
- Example of a schedule that is not conflict serializable:

T_3	T_4
read(Q)	
	write(Q)
write(Q)	

- We are unable to swap instructions in the above schedule to obtain either the serial schedule < T_3, T_4 >, or the serial schedule < T_4, T_3 >.

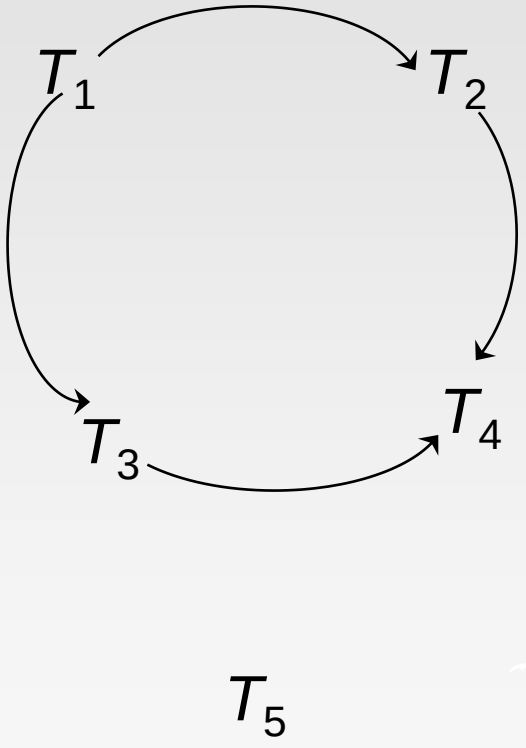
Testing for Serializability

- Consider some schedule of a set of transactions T_1, T_2, ..., T_n
- **Precedence graph** — a direct graph where
  - the vertices are the transactions (names).
  - there is an arc from T_i to T_j if the two transaction conflict, and T_i accessed the data item on which the conflict arose earlier.
- We may label the arc by the item that was accessed.
- **Example 1**



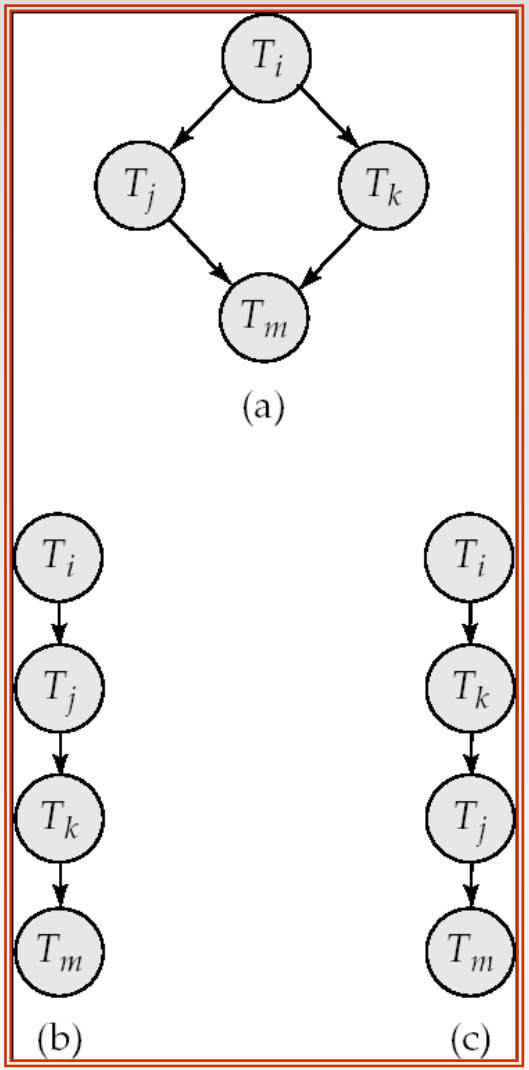
Example Schedule (Schedule A) + Precedence Graph

T_1	T_2	T_3	T_4	T_5
	read(X)			
read(Y)				
read(Z)				
				read(V)
				read(W)
				read(W)
	read(Y)			
	write(Y)			
		write(Z)		
read(U)				
			read(Y)	
			write(Y)	
			read(Z)	
			write(Z)	
read(U)				
write(U)				



Test for Conflict Serializability

- A schedule is conflict serializable if and only if its precedence graph is acyclic.
- Cycle-detection algorithms exist which take order n^2 time, where n is the number of vertices in the graph.
  - (Better algorithms take order $n + e$ where e is the number of edges.)
- If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph.
  - This is a linear order consistent with the partial order of the graph.
  - For example, a serializability order for Schedule A would be $T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4$



View Serializability

- Sometimes it is possible to serialize schedules that are not conflict serializable
- View serializability provides a weaker and still consistency preserving notion of serialization
- Let *S* and *S´* be two schedules with the same set of transactions. *S* and *S´* are **view equivalent** if the following three conditions are met, for each data item *Q,*
  1. If in schedule S, transaction T_i reads the initial value of *Q*, then in schedule *S'* also transaction T_i must read the initial value of *Q.*
  2. If in schedule S transaction T_i executes **read**(*Q)*, and that value was produced by transaction T_j (if any), then in schedule *S'* also transaction T_i must read the value of *Q* that was produced by the same **write**(Q) operation of transaction T_j .
  3. The transaction (if any) that performs the final **write**(*Q*) operation in schedule *S* must also perform the final **write**(*Q*) operation in schedule *S'.*

View Serializability (Cont.)

- A schedule *S* is **view serializable** if it is view equivalent to a serial schedule.
- Every conflict serializable schedule is also view serializable.
- Below is a schedule which is view-serializable but *not* conflict serializable.

T_3	T_4	T_6
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

- It is equivalent to either <T3,T4,T6> or <T4,T3,T6>
- Every view serializable schedule that is not conflict serializable has **blind writes.**

MCQ

Let S be the following schedule of operations of three transactions T_1, T_2 and T_3 in a relational database system:
$R_2(Y), R_1(X), R_3(Z), R_1(Y), W_1(X), R_2(Z), W_2(Y), R_3(X), W_3(Z)$
Consider the statements P and Q below:
P: S is conflict-serializable.
Q: If T_3 commits before T_1 finishes, then S is recoverable.
Which one of the following choices is correct?

1. P is true and Q is false.
2. Both P and Q are true.
3. P is false and Q is true.
4. Both P and Q are false.

Recoverability in DBMS

- As discussed, a transaction may not execute completely due to hardware failure, system crash or software issues.
- In that case, we have to roll back the failed transaction. But some other transaction may also have used values produced by the failed transaction. So we have to roll back those transactions as well.

Recoverable Schedules

T1	T2
R(A) A=A-5 W(A) commit	 R(A) A=A-2 W(A) commit

Schedules in which transactions commit only after all transactions whose changes they read commit are called recoverable schedules. In other words, if some transaction Tj is reading value updated or written by some other transaction Ti, then the commit of Tj must occur after the commit of Ti.

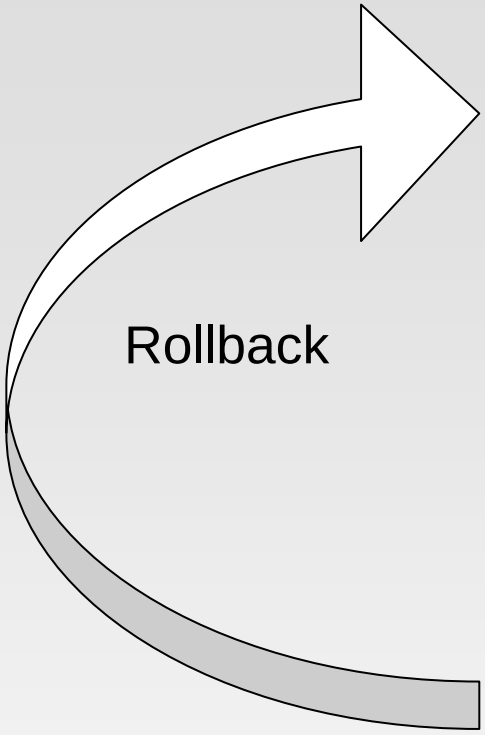
Recoverable schedule — if a transaction T_j reads a data item previously written by a transaction T_i , then the commit operation of T_i must appear before the commit operation of T_j.

Recoverable Schedules

- Example 1:
- S1: R1(x), W1(x), R2(x), R1(y), R2(y), W2(x), W1(y), C1, C2;

T1	T2
R(X)	
W(X)	
	R(X)
R(Y)	
	R(Y)
	W(X)
W(Y)	
C1	
	C2

Irrecoverable Schedules



T1	T2
R(A) A=A-5 W(A)	
	R(A) A=A-2 W(A) commit
R(B) * Fail	

Suppose A=10
A=10-5=5
A=5

A=5
A=5-2=3
A=3

B=20(Suppose)

At this point,Transaction failed due to any reason

A=10 again

Cascading Rollbacks

- **Cascading rollback** – a single transaction failure leads to a series of transaction rollbacks. Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)

T_{10}	T_{11}	T_{12}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)

  If T_{10} fails, T_{11} and T_{12} must also be rolled back.

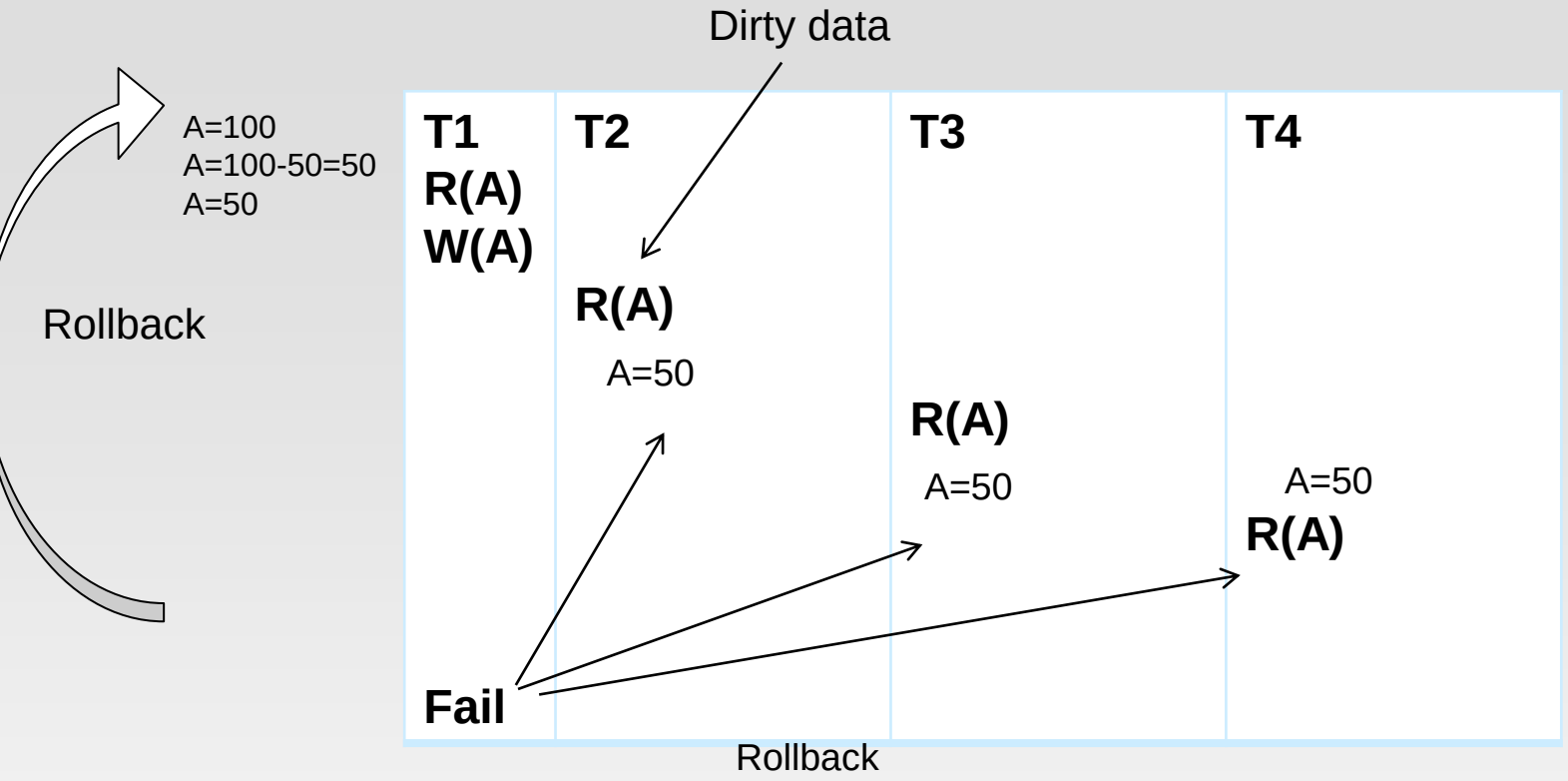
- Can lead to the undoing of a significant amount of work

Disadvantage: CPU cycle gets wasted

Cascading Rollbacks

- The table below shows a schedule with two transactions, T1 reads and writes A and that value is read and written by T2. But later on, T1 fails. So we have to rollback T1. Since T2 has read the value written by T1, it should also be rollbacked. As it has not committed, we can rollback T2 as well. So it is recoverable with cascading rollback. Therefore, if Tj is reading value updated by Ti and commit of Tj is delayed till commit of Ti, the schedule is called recoverable with cascading rollback.

T1	T2
R(A) A=A-5 W(A) Fail Commit	 R(A) A=A+2 W(A) Commit



Cascading Rollback Schedule

Cascadeless Schedules

- **Cascadeless schedules** — in these, cascading rollbacks cannot occur; for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_j.
- Every cascadeless schedule is also recoverable
- It is desirable to restrict the schedules to those that are cascadeless

Cascadeless Recoverable Rollback:

The table below shows a schedule with two transactions, T1 reads and writes A and commits and that value is read by T2. But if T1 fails before commit, no other transaction has read its value, so there is no need to rollback other transaction. So this is a Cascadeless recoverable schedule. So, if Tj reads value updated by Ti only after Ti is committed, the schedule will be cascadeless recoverable.

Cascadless

T1	T2
R(A) A=A-5 W(A) Commit	R(A) A=A+2 W(A) Commit

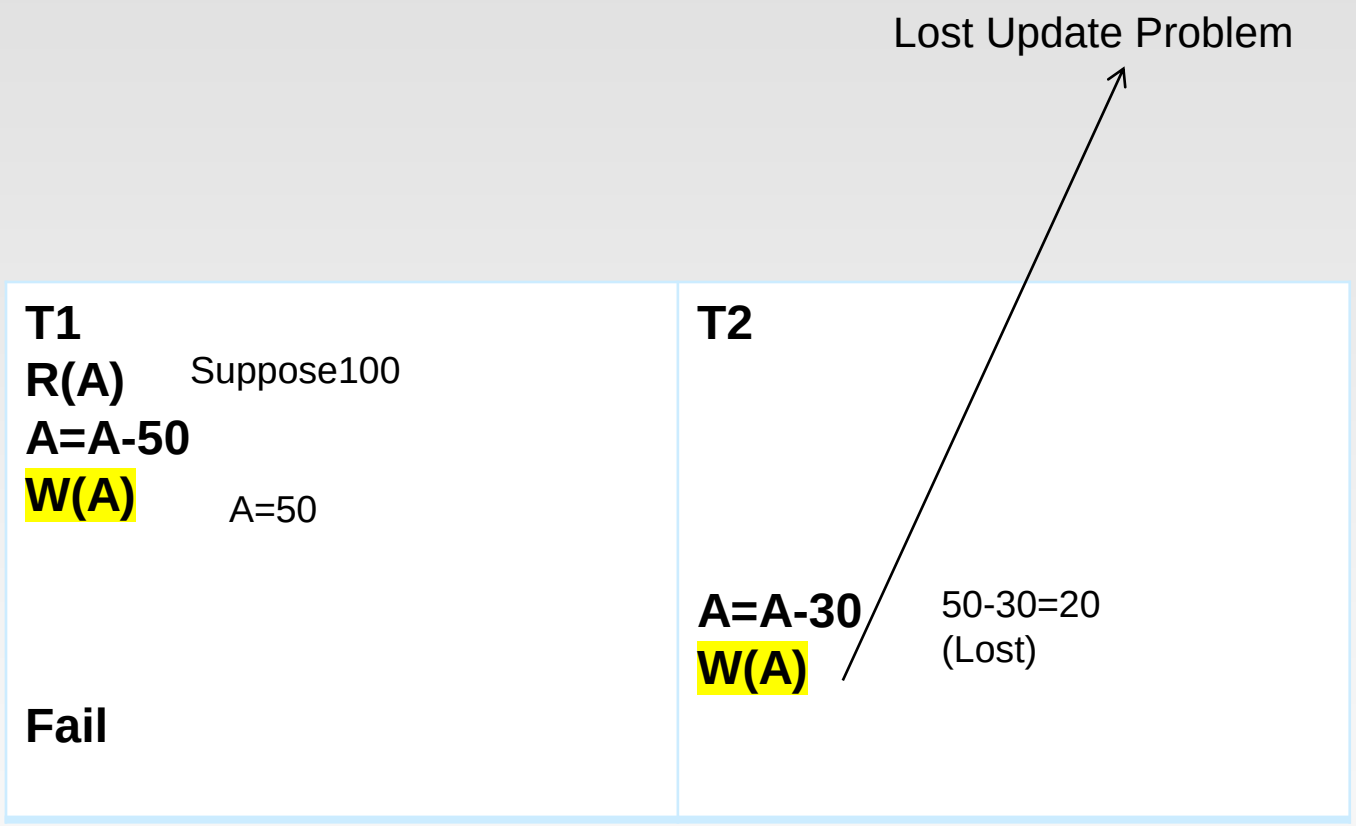
T1	T2
W(A)	
	R(A)

Cascade Rollback Example

T1	T2
W(A) C1	
	R(A)

Cascadeless Schedule Example

Cascadeless Schedule



MCQs

- Which of the following scenarios may lead to an irrecoverable error in a database system?

- A transaction writes a data item after it is read by an uncommitted transaction.
- A transaction reads a data item after it is read by an uncommitted transaction.
- A transaction reads a data item after it is written by a committed transaction.
- A transaction reads a data item after it is written by an uncommitted transaction.

MCQs

Match the following:

P. Recoverable	1. T_j reads data items written by T_i, the T_j commits after T_i commits
Q. Cascadeless	2. Reading uncommitted data
R. Dirty read	3. T_j reads data items written by T_i, the T_i commits after T_j commits
S. Non recoverable	4. T_j reads data items written by T_i, the commit operation of T_i appears before the read operation of T_j

- ◯ P-2, Q-1, R-4, S-3
- ◯ P-2, Q-3, R-4, S-1
- ◯ P-3, Q-4, R-2, S-1
- ◯ P-1, Q-4, R-2, S-3

MCQs

Question-5.

Consider the schedule:

T1: R(X) ; T2: R(Y) ; T3:W(X) ; T2:R(X) ; T1:R(Y)

The schedule T1 is

- ○ Not conflict and not view serializable
- ○ Conflict and view serializable
- ○ Not conflict, but view serializable
- ○ Conflict, but not view serializable

MCQs

Question-6.

Problem of testing view serializability is

- ◯ P Problem
- ◯ NP hard
- ◯ NP problem
- ◯ NP complete

Which of the following statement is not correct for serializability of transactions?

- In a serial schedule, each transaction is independent of others.
- In non-serial schedule, we allow the two transactions to overlap their execution
- A non-serial schedule may not always result in an incorrect outcome.
- Every schedule is serializable.

MCQs

Which type of sorting is performed in precedence graph for serial execution?

- A. Topological
- B. Depth First Search
- C. Breadth First Search
- D. Ascending order of transaction indices

Concurrency Control

- A database must provide a mechanism that will ensure that all possible schedules are
  - either conflict or view serializable, and
  - are recoverable and preferably cascadeless
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency
  - Are serial schedules recoverable/cascadeless?
- Testing a schedule for serializability *after* it has executed is a little too late!
- **Goal** – to develop concurrency control protocols that will assure serializability.
  - Lock-based protocols
  - Timestamp-based protocols

Lock-Based Protocols
Shared-Exclusive Protocol

- A lock is a mechanism to control concurrent access to a data item
- Data items can be locked in two modes :
  1. *exclusive (X) mode*. Data item can be both read as well as written. X-lock is requested using **lock-X** instruction.
  2. *shared (S) mode*. Data item can only be read. S-lock is requested using **lock-S** instruction.
- Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted.

T1	T2
S(A)	X(A)
R(A)	R(A)
U(A)	W(A)
	U(A)

Lock-Based Protocols (Cont.)

- Lock-compatibility matrix

Request

Grant

	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.

Lock Based Compatability Matrix Cases

T1	T2
R(A)	
	R(A)

Case1

T1	T2
R(A)	
	R(A)
	W(A)

Case 2

T1	T2
R(A)	
W(A)	
	R(A)

Case3

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)

Case4

Drawbacks

- May not sufficient to produce only serializable schedule.

T1	T2
X(A)	
R(A)	
W(A)	
U(A)	S(A)
	R(A)
X(A)	U(A)
R(B)	
W(B)	
U(B)	

- May not free from Irrecoverability

T1	T2
X(A)	
R(A)	
W(A)	
U(A)	S(A)
	R(A)
	U(A)
	Commit
Fail	

- May not free from Deadlock

T1	T2
X(A) Grant	
	X(B) Grant
X(B) Wait	
	X(A) Wait

- May not free from Starvation

Starvation

T1	T2	T3	T4
	S(A) Grant		
X(A) Wait	.		
.	.	S(A)	
.	.	.	
	U(A)	.	
	Wait	U(A)	
			S(A) Grant
			.
			.
			U(A)

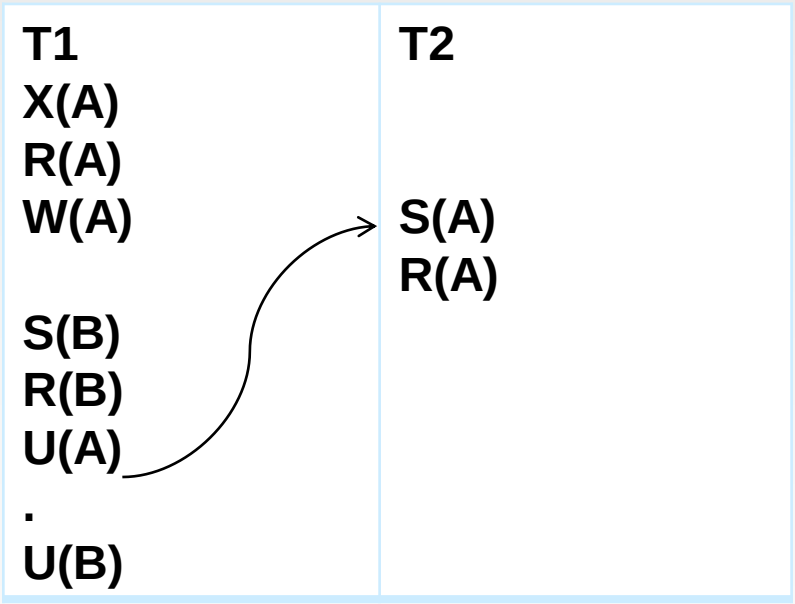
The Two-Phase Locking Protocol

- This is a protocol which ensures conflict-serializable schedules.
- Phase 1: Growing Phase
  - transaction may obtain locks
  - transaction may not release locks
- Phase 2: Shrinking Phase
  - transaction may release locks
  - transaction may not obtain locks
- The protocol assures serializability. It can be proved that the transactions can be serialized in the order of their **lock points** (i.e. the point where a transaction acquired its final lock).

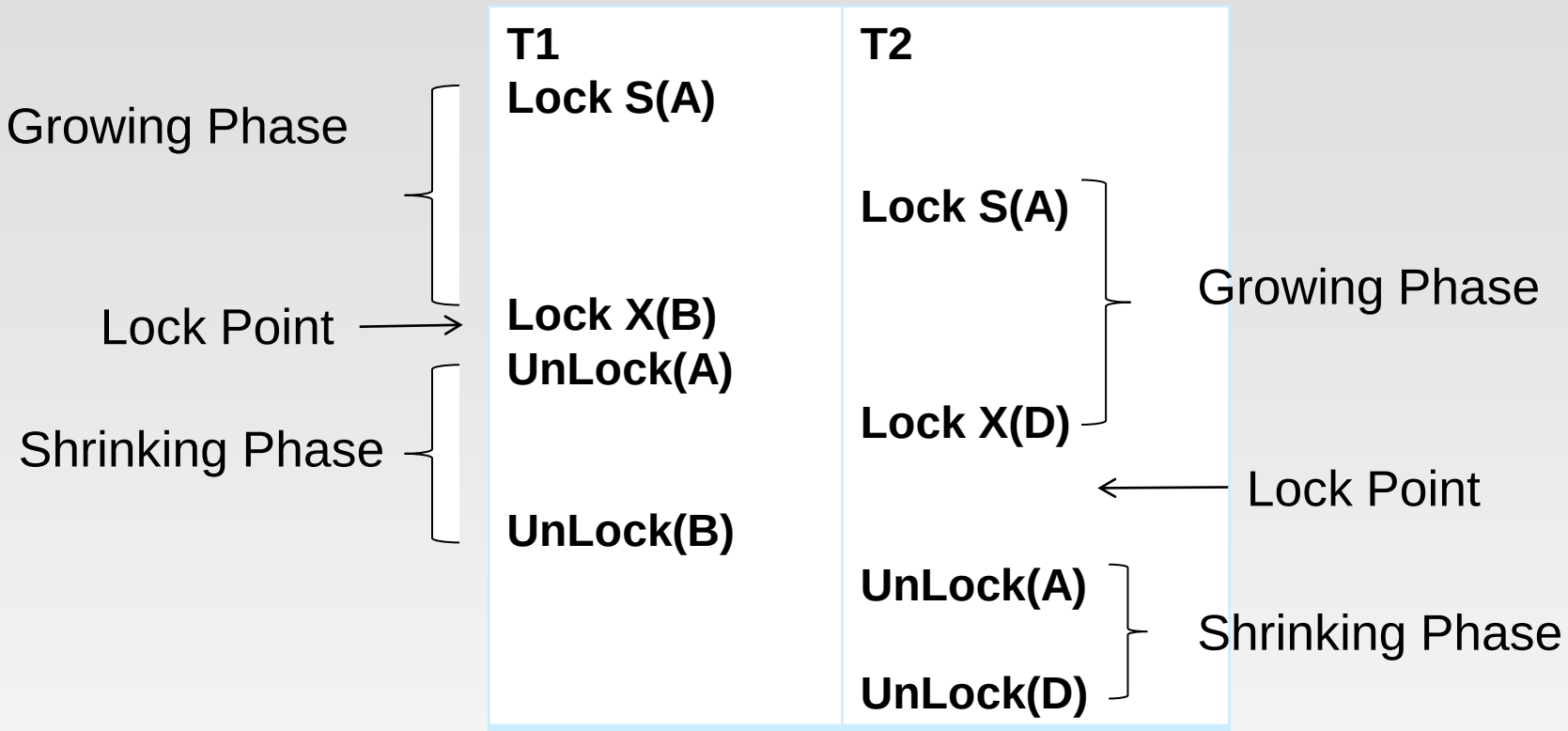
Growing Phase

Shrinking Phase

T1
X(A)
R(A)
W(A)
S(B)
R(B)
U(A)



T1---->T2
Serializability is achieved


T1
Lock S(A)
Lock X(B)
UnLock(A)
UnLock(B)
T2
Lock S(A)
Lock X(D)
UnLock(A)
UnLock(D)
Growing Phase
Lock Point
Shrinking Phase
Growing Phase
Lock Point
Shrinking Phase

Advantage and Pitfalls of Basic Two PL

- Advantage: Always ensures Serializability
- Disadvantage:May not free from Irrecoverability.
  May not free from Deadlock
  May not free from Starvation
  May not free from Cascading Rollbacks

T1	T2
Lock X(A)	
R(A)	
W(A)	
UnLock(A)	
	Lock S(A)
	R(A)
	Commit
Fail	

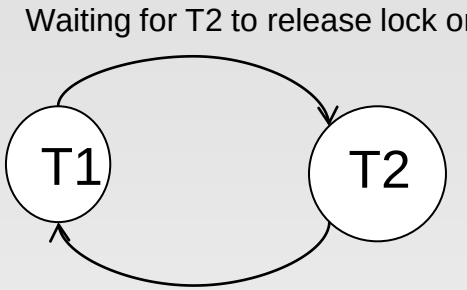
T1 can rollback but T2 can't- Irrecoverable Schedule

T1	T2	T3	T4
X(A)			
R(A)			
W(A)	R(A)		
		R(A)	
			R(A)
Fail			

Cascading Schedule degrade performance

Deadlock

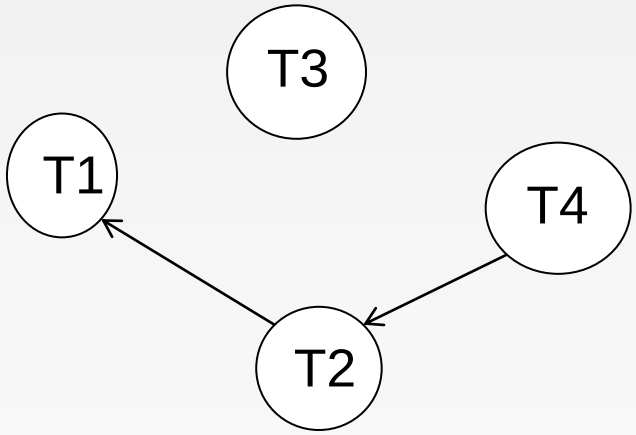
- A system is in deadlock state if there exists a set of transactions such that every transaction in the set is waiting for another transaction in the set.



X,Y	X,Y
Lock(X)	Lock(Y)

- Draw Wait for Graph G=(V,E) to detect deadlock where V=nodes describing Transactions and E is directed edge
- Ti--->Tj(directed edge){Ti is waiting for data item held by Tj.

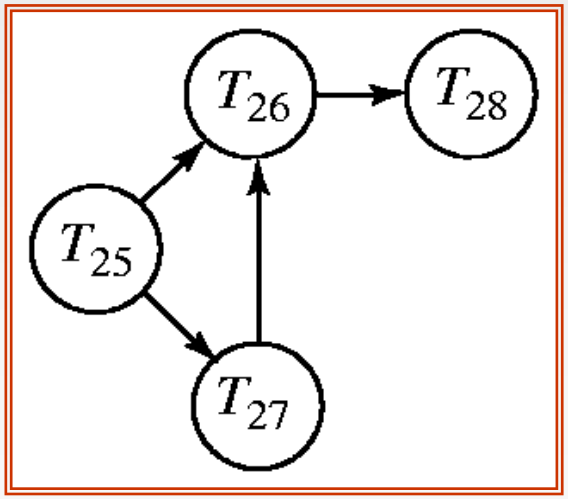
Transactions	Data items	Locks
T1	Q	Shared
T2	P	Exclusive
	Q	Exclusive
T3	Q	Shared
T4	P	Exclusive



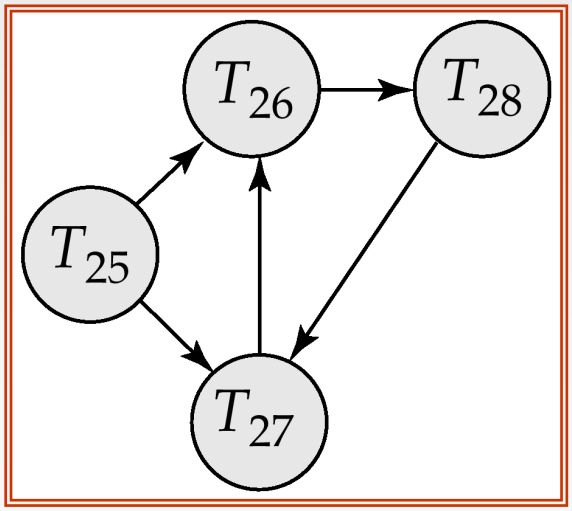
If cycle is present in Wait for Graph, system is in deadlock state. Here, no cycle, so no deadlock.

Deadlock Detection

- Deadlocks can be described as a *wait-for graph* where:
  - vertices are all the transactions in the system
  - There is an edge $T_i \rightarrow T_k$ in case T_i is waiting for T_k
- When T_i requests a data item currently being held by T_k, then the edge $T_i \rightarrow T_k$ is inserted in the wait-for graph. This edge is removed only when T_k is no longer holding a data item needed by T_i.
- The system is in a deadlock state if and only if the wait-for graph has a cycle. Must invoke a deadlock-detection algorithm periodically to look for cycles.



Wait-for graph without a cycle



Wait-for graph with a cycle

Necesssary Conditions for Deadlock

- Mutual Exclusive
- Hold and Wait
- No preemption
- Circular Wait

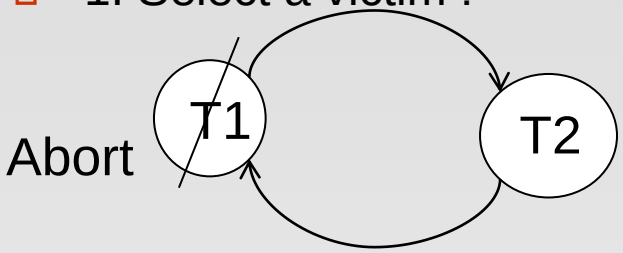
Deadlock Prevention

- Use of Timestamp: Ti request for data item held by Tj
- Wait Die: If Ts(Ti)<Ts(Tj) [Ti is older than Tj], Ti is allowed to wait otherwise if Ti is younger than Tj then abort Ti(Ti dies) and restart it later with same time stamp.

- Wound wait:If Ts(Ti)<Ts(Tj) [Ti is older than Tj], then abort Tj(Ti wounds Tj and restart it with same timestamp.If Ts(Ti)>Ts(Tj),then Ti is allowed to wait.
- Time Out Based Scheme- If Transaction that has requested a lock waits for at most a specified amount of time.If the lock is not granted within that time,transaction is said to timeout and it rollsback and restarts.

Deadlock Recovery

- 1. Select a victim :



- Guidelines for selecting victim:(of minimum cost)
- - Length of Transaction(Younger)
- - Data item used by transaction(less number of data item)
- - Data item that are to be locked(more data items to lock)
- -How man transactions to rollback(minimum)
- 2. Rollback--- 1. Full 2. Partial
- 3. Starvation

Modified Two-Phase Locking Protocol

Strict two-phase locking

- Cascading roll-back is possible under two-phase locking. To avoid this, follow a modified protocol called **strict two-phase locking**. Here a transaction must hold all its exclusive locks till it commits/aborts.

T1	T2
X(A)	
R(A)	
W(A)	R(A)
C	
U(A)	
Fail	

Cascading Removed

T1	T2
X(A)	
R(A)	
W(A)	R(A)
	C
C	
U(A)	
Fail	

Irrecoverability removed

Rigorous two-phase locking

- **Rigorous two-phase locking** is even stricter: here *all* locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit.

Note: Cascadeless and Strict recoverable Schedules are achieved from this modified 2 PL.

Timestamp-Based Protocols

- Instead of determining the order of each operation in a transaction at execution time, determines the order by the time of beginning of each transaction.
  - Each transaction is issued a timestamp when it enters the system. If an old transaction T_i has time-stamp TS(T_i), a new transaction T_j is assigned time-stamp TS(T_j) such that TS(T_i) <TS(T_j).
  - The protocol manages concurrent execution such that the time-stamps determine the serializability order.
- In order to assure such behavior, the protocol maintains for each data Q two timestamp values:
  - **W-timestamp**(Q) is the largest time-stamp of any transaction that executed **write**(Q) successfully.
  - **R-timestamp**(Q) is the largest time-stamp of any transaction that executed **read**(Q) successfully.

TimeStamp Ordering Protocol

- Unique Value Assigned to every Transaction
- Tells the order in which they enter into the system
- Read_TS(RTS)-->Last(latest) transaction number which performed read operation successfully.
- Write_TS(WTS)--> Last(latest) transaction number which performed Write operation successfully.
- Rules:

1. Transaction Ti issues a R(A) operation:

a) if WTS(A)>TS(Ti), Rollback Ti

b) Otherwise execute R(A) operation

Set RTS(A)= Max{ RTS(A),TS(Ti)}

2. Transaction Ti issues a W(A) operation:

a) if RTS(A)>TS(Ti), Rollback Ti

b) if WTS(A)>TS(Ti), Rollback Ti

c)Otherwise execute W(A) operation

Set WTS(A)=TS(Ti)

T1(OLDER)100 **R(A)**	**T2(YOUNGER)200** **W(A)**

Ts(Ti)=Timestamp of Ti

T1(10) **W(A)**	**T2(20)** **W(A)**	**T3(30)** **W(A)**

Allowed Cases

T1(OLDER)100	T2(YOUNGER) 200
R(A)	
	W(A)

T1(OLDER)100	T2(YOUNGER) 200
W(A)	
	R(A)

T1(OLDER)100	T2(YOUNGER)200
W(A)	
	W(A)

Not Allowed Cases

T1(OLDER)100	T2(YOUNGER) 200
	W(A)
R(A)	

T1(OLDER)100	T2(YOUNGER) 200
	R(A)
W(A)	

T1(OLDER)100	T2(YOUNGER) 200
	W(A)
W(A)	

Which transaction is rolling Back?

100 200 300

T1(Oldest)	T2	T3(Youngest)
R(A)		
	R(B)	
W(C)		
		R(B)
R(C)		
	W(B)	
		W(A)

Rules:

1. Transaction Ti issues a R(A) operation:
   a) if WTS(A)>TS(Ti), Rollback Ti
   b) Otherwise execute R(A) operation
      Set RTS(A)= Max{ RTS(A),TS(Ti)}
2. Transaction Ti issues a W(A) operation:
   a) if RTS(A)>TS(Ti), Rollback Ti
   b) if WTS(A)>TS(Ti), Rollback Ti
   c)Otherwise execute W(A) operation
   Set WTS(A)=TS(Ti)

	A	B	C
RTS	0-->100	0-->200-->300	0-->100
WTS	0->300	0	0-->100

Timestamp-Based Protocols (Cont.)

- The timestamp ordering protocol ensures that any conflicting **read** and **write** operations are executed in timestamp order.
- Suppose a transaction T_i issues a **read**(Q)
  1. If TS(T_i) ≤ **W**-timestamp(Q), then T_i needs to read a value of Q that was already overwritten.
     - Hence, the **read** operation is rejected, and T_i is rolled back.
  2. If TS(T_i)≥ **W**-timestamp(Q), then the **read** operation is executed, and **R**-timestamp(Q) is set to **max**(R-timestamp(Q), TS(T_i)).
- Suppose that transaction T_i issues **write**(Q).
  1. If TS(T_i) < **R**-timestamp(Q), then the value of Q that T_i is producing was needed previously, and the system assumed that that value would never be produced.
     - Hence, the **write** operation is rejected, and T_i is rolled back.
  2. If TS(T_i) < **W**-timestamp(Q), then T_i is attempting to write an obsolete value of Q.
     - Hence, this **write** operation is rejected, and T_i is rolled back.
  3. Otherwise, the **write** operation is executed, and **W**-timestamp(Q) is set to TS(T_i).

Correctness of Timestamp-Ordering Protocol

- The timestamp-ordering protocol guarantees serializability since all the arcs in the precedence graph are of the form:

Thus, there will be no cycles in the precedence graph

- Timestamp protocol ensures freedom from deadlock as no transaction ever waits.
- But the schedule may not be cascade-free, and may not even be recoverable.

Recovery Schemes

- Recovery schemes are techniques to ensure database consistency and transaction atomicity and durability despite failures such as transaction failures, system crashes, disk failures.
  - We just briefly focus this issue, which strongly relies on lower-level control (usage of RAID, buffer management)
  - More on this can be found in chapter 17 of the book
- Recovery algorithms have two parts
  1. Actions taken during normal transaction processing to ensure enough information exists to recover from failures
  2. Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability

Recovery and Atomicity

- Modifying the database without ensuring that the transaction commits may leave the database in an inconsistent state.
  - Consider again the transaction T_i that transfers €50 from account *A* to account *B*.
  - Several output operations are required for T_i (to output *A* and *B*). A failure may occur after one of these modifications have been made but before all of them are made.
- To ensure atomicity despite failures, first output information describing the modifications to stable storage (i.e. storage guaranteed/assumed not to fail, e.g. with RAID) without modifying the database itself.
- Two approaches are possible:
  - **log-based recovery**, and
  - **shadow-paging**

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, and V_2 is the value to be written to X.
  - Log record notes that T_i has performed a write on data item X_j X_j had value V_1 before the write, and will have value V_2 after the write.
- When T_i finishes it last statement, the log record <T_i **commi**t> is written.
- For writing the actual records
  - Deferred database modification
  - Immediate database modification

Deferred Database Modification

- The **deferred database modification** scheme records all modifications to the log, and defers **write**s to after partial commit.
- Transaction starts by writing <T_i ***start***> record to log.
- A **write**(X) operation results in a log record <T_i, X, V> being written, where V is the new value for X (old value is not needed).
  - The write is not performed on X at this time, but is deferred.
- When T_i partially commits, <T_i **commit**> is written to the log
- After that, the log records are read and used to actually execute the previously deferred writes.
- During recovery after a crash, a transaction needs to be redone if and only if both <T_i **start**> and<T_i **commit**> are there in the log.
- Redoing a transaction T_i (**redo** T_i) sets the value of all data items updated by the transaction to the new values.

Deffered(No Undo/Redo Operation)

T1 **R(A)** **A=A+100** **W(A)** **R(B)** **B=B+200** **W(B)**	**<T1,Start>** **<T1,A,200>** **<T1,B,400>**

A=100--->200
B=200--->400

<T1,Start>
<T1,A,200>
<T1,B,400>
<T1,commit>
<T2,Start>
<T2,C,500>

Immediate Database Modification

- The **immediate database modification** scheme allows database updates of an uncommitted transaction to be made as the writes are issued
  - since undoing may be needed, update logs must have both old value and new value
- Update log record must be written *before* database item is written
  - We assume that the log record is output directly to stable storage
  - Can be extended to postpone log record output, so long as prior to execution of an **output**(*B*) operation for a data block B, all log records corresponding to items *B* must be flushed to stable storage
- Output of updated blocks can take place at any time before or after transaction commit
- Order in which blocks are output can be different from the order in which they are written.

Immediate Database Modification (Cont.)

- Recovery procedure has two operations instead of one:
  - **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
- Both operations must be **idempotent**
  - That is, even if the operation is executed multiple times the effect is the same as if it is executed once
    - Needed since operations may get re-executed during recovery
- When recovering after failure:
  - Transaction T_i needs to be undone if the log contains the record <T_i **start**>, but does not contain the record <T_i **commit**>.
  - Transaction T_i needs to be redone if the log contains both the record <T_i **start**> and the record <T_i **commit**>.
- Undo operations are performed first, then redo operations.

T1 **R(A)** **A=A+100** **W(A)** **R(B)** **B=B+200** **W(B)**	**<T1,Start>** **<T1,A,100,200>** **<T1,B,200,400>**

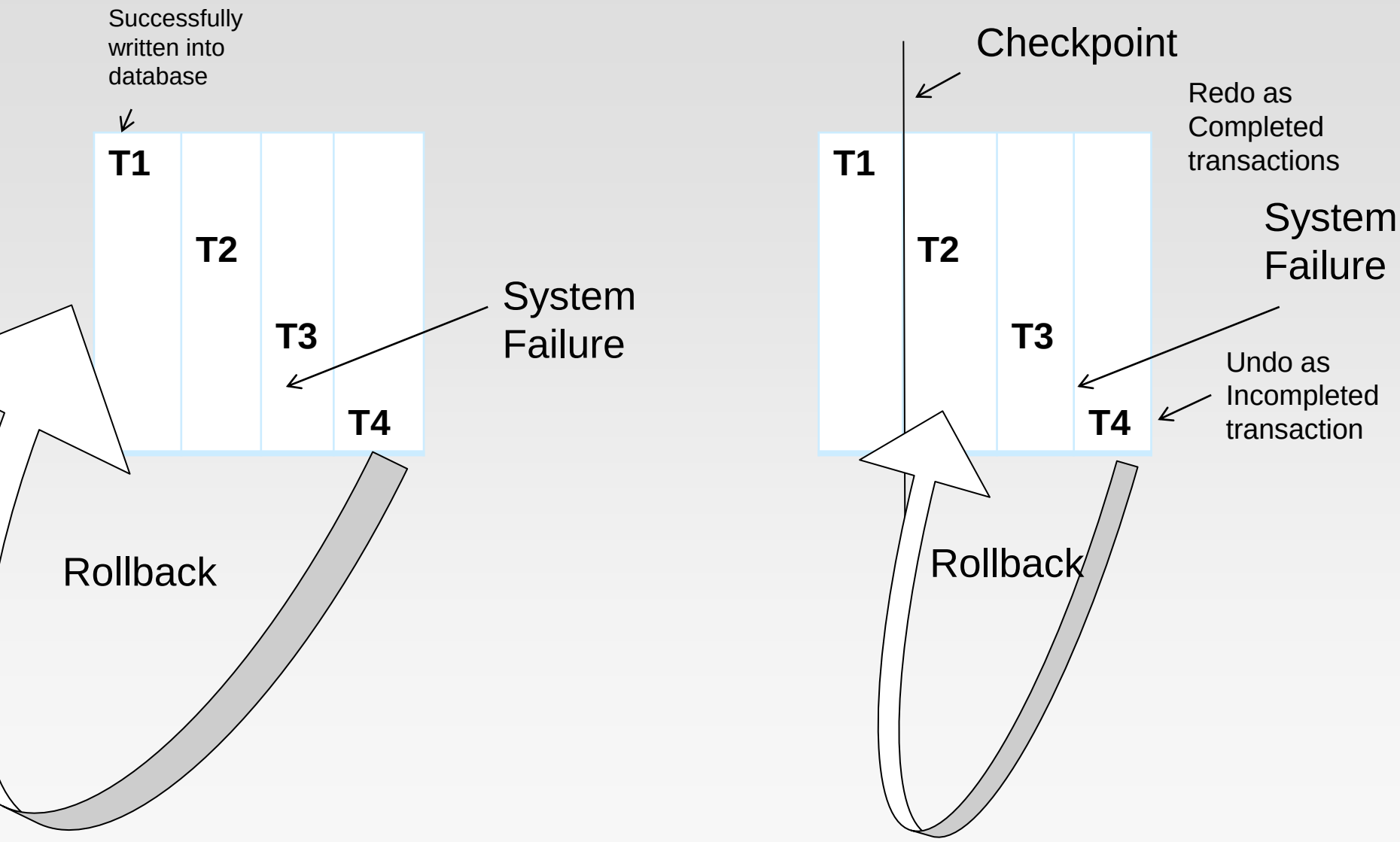
A=100--->200
B=200--->400

Checkpoints

- Problems in recovery procedure as discussed earlier :
  1. searching the entire log is time-consuming
  2. one might unnecessarily redo transactions which have already output their updates to the database.
- 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**> onto stable storage.

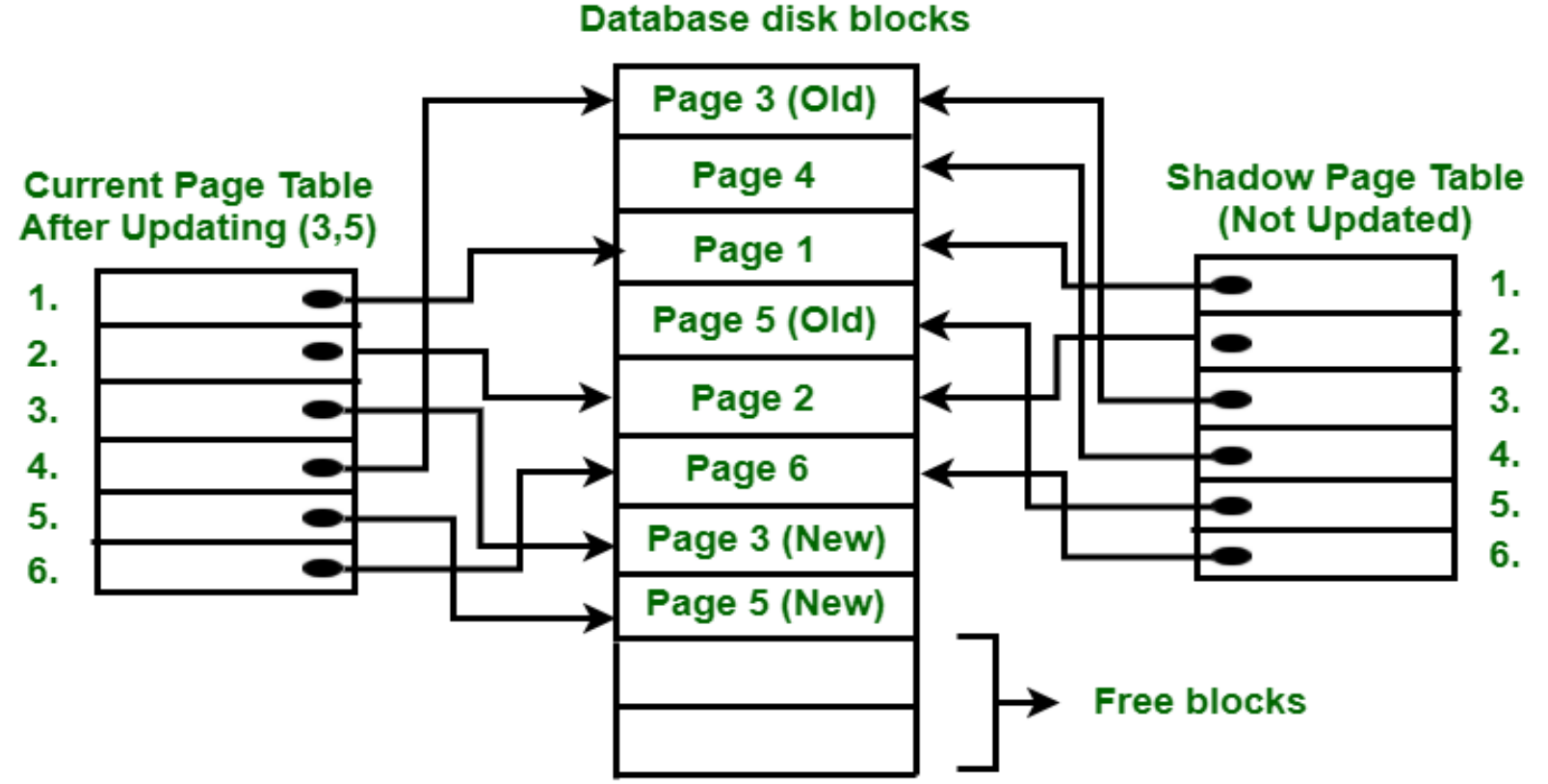
  Advantage: Reduces the amount of time required during recovery.

Checkpoint


Successfully written into database
T1
T2
T3
T4
System Failure
Rollback
Checkpoint
Redo as Completed transactions
T1
T2
T3
T4
System Failure
Undo as Incompleted transaction
Rollback

Shadow Paging

- Shadow Paging is recovery technique that is used to recover database.
- In this recovery technique, database is considered as made up of fixed size of logical units of storage which are referred as pages.
- pages are mapped into physical blocks of storage, with help of the page table which allow one entry for each logical page of database.
- This method uses two page tables named **current page table** and **shadow page table.**
- The entries which are present in current page table are used to point to most recent database pages on disk.
- Another table i.e., Shadow page table is used when the transaction starts which is copying current page table. After this, shadow page table gets saved on disk and current page table is going to be used for transaction.
- Entries present in current page table may be changed during execution but in shadow page table it never get changed. After transaction, both tables become identical.
- This technique is also known as **Cut-of-Place updating.**



Step1: Construct a directory with every entry in the directory pointing to database block
Step2: Create a shadow copy of Directory
step3: Transaction should modify only pages pointed by current directory and write it to a new location on disk
Step 4: If transaction commits, discard shadow directory and old pages.
Step5: If transaction fails, discard current directory and modified pages.

- **COMMIT Operation :**
- To commit transaction following steps should be done :

- All the modifications which are done by transaction which are present in buffers are transferred to physical database.
- Output current page table to disk.
- Disk address of current page table output to fixed location which is in stable storage containing address of shadow page table. This operation overwrites address of old shadow page table. With this current page table becomes same as shadow page table and transaction is committed.
- **Failure :**
- If system crashes during execution of transaction but before commit operation, With this, it is sufficient only to free modified database pages and discard current page table. Before execution of transaction, state of database get recovered by reinstalling shadow page table.
- If the crash of system occur after last write operation then it does not affect propagation of changes that are made by transaction. These changes are preserved and there is no need to perform redo operation.

Advantages :

- This method require fewer disk accesses to perform operation.
- In this method, recovery from crash is inexpensive and quite fast.
- There is no need of operations like- Undo and Redo.

Disadvantages :

- Due to location change on disk due to update database it is quite difficult to keep related pages in database closer on disk.
- During commit operation, changed blocks are going to be pointed by shadow page table which have to be returned to collection of free blocks otherwise they become accessible.
- The commit of single transaction requires multiple blocks which decreases execution speed.
- To allow this technique to multiple transactions concurrently it is difficult.

Transaction Definition in SQL

- Data manipulation language must include a construct for specifying the set of actions that comprise a transaction.
- In SQL, a transaction begins implicitly, after previous transaction.
- A transaction in SQL ends by:
  - **Commit work** commits current transaction and begins a new one.
  - **Rollback work** causes current transaction to abort.
- In almost all database systems, by default, every SQL statement also commits implicitly if it executes successfully
  - Implicit commit can be turned off by a database directive
    - E.g. in JDBC, connection.setAutoCommit(false);
- Four levels of (weak) consistency, cf. before.

Transaction management in Oracle

- Transaction beginning and ending as in SQL
  - Explicit **commit work** and **rollback work**
  - Implicit commit on session end, and implicit rollback on failure
- Log-based deferred recovery using rollback segment
- Checkpoints (inside transactions) can be handled explicitly
  - **savepoint** <name>
  - **rollback to** <name>

- Concurrency control is made by (a variant of) multiversion rigorous two-phase locking
- Deadlock are detected using a *wait-graph*
  - Upon deadlock detection, the last transaction that detects the deadlock is rolled back

Levels of Consistency in Oracle

- Oracle implements 2 of the 4 of levels of SQL
  - *Read committed*, by default in Oracle and with
    - **set transaction isolation level read committed**
  - *Serializable*, with
    - **set transaction isolation level serializable**
    - Appropriate for large databases with only few updates, and usually with not many conflicts. Otherwise it is too costly.
- Further, it supports a level similar to *repeatable read*:
  - Read only mode, only allow reads on committed data, and further doesn't allow INSERT, UPDATE or DELETE on that data. (without unrepeatable reads!)
    - **set transaction isolation level read only**

Granularity in Oracle

- By default Oracle performs **row level locking**.
- Command

  select … for update

  locks the selected rows so that other users cannot lock or update the rows until you end your transaction. Restriction:
  - Only at top-level select (not in sub-queries)
  - Not possible with **DISTINCT** operator, **CURSOR** expression, set operators, **group by** clause, or aggregate functions.
- Explicit locking of tables is possible in several modes, with
  - **lock table** <name> **in**
    - **row share mode**
    - **row exclusive mode**
    - **share mode**
    - **share row exclusive mode**
    - **exclusive mode**

Lock modes in Oracle

- Row share mode
  - The least restrictive mode (with highest degree of concurrency)
  - Allows other transactions to query, insert, update, delete, or lock rows concurrently in the same table, except for exclusive mode
- Row exclusive mode
  - As before, but doesn't allow setting other modes except for row share.
  - Acquired automatically after a **insert**, **update** or **delete** command on a table
- Exclusive mode
  - Only allows queries to records of the locked table
  - No modifications are allowed
  - No other transaction can lock the table in any other mode
- See manual for details of other (intermediate) modes

Consistency tests in Oracle

- By default, in Oracle all consistency tests are made immediately after each DML command (insert, delete or update).
- However, it is possible to defer consistency checking of constraints (primary keys, candidate keys, foreign keys, and check conditions) to the end of transactions.
  - Only this makes it possible e.g. to insert tuples in relation with *circular* dependencies in foreign keys
- For this:
  - each constraints that may possibly be deferred must be declared as deferrable:
    - At the definition of the constraint add **deferrable** immediately afterwards
  - at the transaction in which one wants to defer the verification of the constraints, add command:
    - **set constraints all deferred**
    - In this command, instead of **all** it is possible to specify which constraints are to be deferred, by giving their names separated by commas

PL/SQL

Introduction

- Pl/SQL stands for "Procedural Language extension of SQL" that is used in Oracle. PL/SQL is integrated with Oracle database (since version 7).
- It was developed by Oracle Corporation in the late 1980s to enhance the capabilities of SQL.
- PL/SQL is a block structured language. The programs of PL/SQL are logical blocks that can contain any number of nested sub-blocks.
- The functionalities of PL/SQL usually extended after each release of Oracle database. Although PL/SQL is closely integrated with SQL language, yet it adds some programming constraints that are not available in SQL.

Basic Syntax of PL/SQL

- Basic Syntax of PL/SQL is discussed here,which is a block-structured language; this means that the PL/SQL programs are divided and written in logical blocks of code. Each block consists of three sub-parts −
- **Sections & Description**
- **1.Declarations**

This section starts with the keyword DECLARE. It is an optional section and defines all variables, cursors, subprograms, and other elements to be used in the program.

- **2.Executable Commands**

This section is enclosed between the keywords BEGIN and END and it is a mandatory section. It consists of the executable PL/SQL statements of the program. It should have at least one executable line of code, which may be just a NULL command to indicate that nothing should be executed.

- **3.Exception Handling**
- This section starts with the keyword EXCEPTION. This optional section contains exception(s) that handle errors in the program.

Environment Setup of PL/SQL

- PL/SQL is not a standalone programming language; it is a tool within the Oracle programming environment.
- SQL* Plus is an interactive tool that allows you to type SQL and PL/SQL statements at the command prompt. These commands are then sent to the database for processing. Once the statements are processed, the results are sent back and displayed on screen.
- To run PL/SQL programs, you should have the Oracle RDBMS Server installed in your machine. This will take care of the execution of the SQL commands. The most recent version of Oracle RDBMS is 11g.

Basic Syntax of PL/SQL

- Every PL/SQL statement ends with a semicolon (;).
- PL/SQL blocks can be nested within other PL/SQL blocks using BEGIN and END. Following is the basic structure of a PL/SQL block −

```
DECLARE
<declarations section>
BEGIN
<executable command(s)>
EXCEPTION
<exception handling>
END;
```

Advantages of PL/SQL:

- 1. PL/SQL is a procedural language.
- 2. PL/SQL is a block structure language.
- 3. PL/SQL handles the exceptions.
- 4. PL/SQL engine can process the multiple SQL statements simultaneously as a single block hence reduce network traffic and provides better performance.

- **The 'Hello World' Example**

```
DECLARE
message  varchar2(20):= 'Hello, World!';
BEGIN
dbms_output.put_line(message);
END;
/
```

- The end; line signals the end of the PL/SQL block. To run the code from the SQL command line, you may need to type / at the beginning of the first blank line after the last line of the code.

The PL/SQL Identifiers

- PL/SQL identifiers are constants, variables, exceptions, procedures, cursors, and reserved words. The identifiers consist of a letter optionally followed by more letters, numerals, dollar signs, underscores, and number signs and should not exceed 30 characters.

- By default, identifiers are not case-sensitive. So you can use integer or INTEGER to represent a numeric value. You cannot use a reserved keyword as an identifier.

PL/SQL Delimiters

- A delimiter is a symbol with a special meaning. Following is the list of delimiters in PL/SQL −

Delimiter	Description
+, -, *, /	Addition, subtraction/negation, multiplication, division
%	Attribute indicator
'	Character string delimiter
.	Component selector
(,)	Expression or list delimiter
:	Host variable indicator
,	Item separator
"	Quoted identifier delimiter
=	Relational operator
@	Remote access indicator
;	Statement terminator

PL/SQL Delimiters

- := Assignment operator
- => Association operator
- || Concatenation operator
- ** Exponentiation operator
- <<, >> Label delimiter (begin and end)
- /*, */ Multi-line comment delimiter (begin and end)
- -- Single-line comment indicator
- .. Range operator
- <, >, <=, >= Relational operators
- <>, '=, ~=, ^= Different versions of NOT EQUAL

PL/SQL Comments

- Program comments are explanatory statements that can be included in the PL/SQL code that you write and helps anyone reading its source code. All programming languages allow some form of comments.

- The PL/SQL supports single-line and multi-line comments. All characters available inside any comment are ignored by the PL/SQL compiler. **The PL/SQL single-line comments start with the delimiter -- (double hyphen) and multi-line comments are enclosed by /* and */.**

Comments

```
DECLARE
-- variable declaration
message  varchar2(20):= 'Hello, World!';
BEGIN
   /*
   *  PL/SQL
      executable statement(s)
  */
 dbms_output.put_line(message);
END;
/
```

PL/SQL Program Units

A PL/SQL unit is any one of the following −

- PL/SQL block
- Function
- Package
- Package body
- Procedure
- Trigger
- Type
- Type body

Variable

Variable is the name of reserved memory location. Each variable has a specific data type which determines the range of values and set of operations for that variable.

- PL/SQL variables naming rules:
- A variable name can’t contain more than 30 characters.
- A variable name must start with an ASCII letter followed by any number, underscore (_) or dollar sign ($).
- PL/SQL is case-insensitive i.e. var and VAR refer to the same variable.

How to declare variable in PL/SQL:

- We have to declare a PL/SQL variable in the declaration section or in a package as a global variable. After declaration PL/SQL allocates memory for the variable and variable name is used to identify the storage location.

Declaring variable in PL/SQL

- **Syntax:**
- variable_name [CONSTANT] datatype [NOT NULL] [:= | DEFAULT initial_value]

Where:

variable_name is a valid identifier name.

datatype is a valid PL/SQL datatype.

- **Initializing Variables in PL/SQL:**

When we declare a variable PL/SQL assigns it NULL as default value. If we want to initialize a variable with a non-NULL value, we can do it during the declaration. We can use any one of the following methods:

- 1. The assignment operator

Num1 binary_integer := 0;

- 2. The DEFAULT keyword

siteName varchar2(20) DEFAULT ‘w3spoint’;

```
DECLARE
var1 integer := 20;
var2 integer := 40;
var3 integer;
var4 real;
BEGIN
var3 := var1 + var2;
dbms_output.put_line('Value of var3: ' || var3);
var4 := 50.0/3.0;
dbms_output.put_line('Value of var4: ' || var4);
END;
/
```

Output

```
Value of var3: 60
Value of var4: 16.6666666666666666666666666666666666666667
```

Variable Scope in PL/SQL:

- PL/SQL allows the nesting of blocks i.e. blocks with blocks. Based on the nesting structure PL/SQL variables can be divide into following categories:
- **Local variables –** Those variables which are declared in an inner block and not accessible to outer blocks are known as local variables.
- **Global variables –** Those variables which are declared in the outer block or a package and accessible to itself and inner blocks are known as global variables.

```
 -- Global variables
num1 number := 10;
num2 number := 20;
BEGIN
dbms_output.put_line('Outer Variable num1: ' || num1);
dbms_output.put_line('Outer Variable num2: ' || num2);
DECLARE
-- Local variables
num3 number := 30;
num4 number := 40;
BEGIN
dbms_output.put_line('Outer variable in inner block num1: ' || num1);
dbms_output.put_line('Outer variable in inner block num2: ' || num2);
dbms_output.put_line('Inner Variable num3: ' || num3);
dbms_output.put_line('Inner Variable num4: ' || num4);
END;
    END;
/
```

- When you declare a PL/SQL variable to hold the column values, it must be of correct data types and precision, otherwise error will occur on execution.
- Rather than hard coding the data type and precision of a variable. PL/SQL provides the facility to declare a variable without having to specify a particular data type using %TYPE and %ROWTYPE attributes.
- These two attributes allow us to specify a variable and have that variable data type be defined by a table/view column or a PL/SQL package variable.
- A % sign servers as the attribute indicator. This method of declaring variables has an advantage as the user is not concerned with writing and maintaining code.

Following are the types of Variable Attributes in PL/SQL:

- **%TYPE**:
- The %TYPE attribute is used to declare variables according to the already declared variable or database column. It is used when you are declaring an individual variable, not a record.
- The data type and precision of the variable declared using %TYPE attribute is the same as that of the column that is referred from a given table.
- This is particularly useful when declaring variables that will hold database values. When using the %TYPE keyword, the name of the columns and the table to which the variable will correspond must be known to the user.
- These are then prefixed with the variable name. If some previously declared variable is referred then prefix that variable name to the %TYPE attribute.
- The syntax for declaring a variable with %TYPE is:

- <var_name> <tab_name>.<column_name>%TYPE;
- Where <column_name> is the column defined in the <tab_name>.

Consider a declaration.

SALARY EMP.SAL % TYPE;

- This declaration will declare a variable SALARY that has the same data type as column SAL of the EMP table.
- Example:

```
DECLARE
SALARY EMP.SAL % TYPE;
BEGIN
Select SAL into SALARY from EMP where EMPNO = 1;
dbms_output.put_line('Salary of EMPNO1: ‘ || SALARY);
END;
```

- After the execution, this will produce the following result:
- Salary of EMPNO1: = 1600
- PL/SQL procedure successfully completed.

3. A developer would like to use referential datatype declaration on a variable. The variable name is EMPLOYEE_LASTNAME, and the corresponding table and column is EMPLOYEE, and LNAME, respectively. How would the developer define this variable using referential datatypes?

A. Use employee.lname%type.

B. Use employee.lname%rowtype.

C. Look up datatype for EMPLOYEE column on LASTNAME table and use that.

D. Declare it to be type LONG.

- **%ROWTYPE:**
- The %ROWTYPE attribute is used to declare a record type that represents a row in a table. The record can store an entire row or some specific data selected from the table. A column in a row and corresponding fields in a record have the same name and data types.
- The syntax for declaring a variable with %ROWTYPE is:

<var_name> <tab_name>%ROWTYPE;

Where <variable_name> is the variable defined in the <tab_name>.

- Consider a declaration.

EMPLOYEE EMP% ROW TYPE;

- This declaration will declare a record named EMPLOYEE having fields with the same name and data types as that of columns in the EMP table. You can access the elements of EMPLOYEE record as
- EMPLOYEE.SAL := 10000;
- EMPLOYEE.ENAME := ‘KIRAN’;

%ROWTYPE Example

- Example:

```
DECLARE
v_emp employees% ROW TYPE;
BEGIN
  Select * into v_emp FROM employees where employee_id=200;
  dbms_output.put_line(v_emp.first_name ||‘  ‘|| v_emp.salary);
END;
```

After the execution, this will produce the following result:

- PL/SQL procedure successfully completed.
- Ria 40000

Advantages:

- If you don’t know the data type at the time of declaration. The data type assigned to the associated variables will be determined dynamically at run time.
- If the data type of the variable you are referencing changes the %TYPE or %ROWTYPE variable changes at run time without having to rewrite variable declarations. For example: if the ENAME column of an EMP table is changed from a VARCHAR2(10) to VRACHAR2(15) then you don’t need to modify the PL/SQL code.

PL/SQL Constants

A constant holds a value used in a PL/SQL block that does not change throughout the program. It is a user-defined literal value.

- Syntax to declare a constant:

  constant_name CONSTANT datatype := VALUE;

Where:

constant_name – is a valid identifier name.

CONSTANT – is a keyword.

VALUE – is a value which must be assigned to a constant when it is declared. You cannot assign a value later.

```
DECLARE
    -- constant declaration
    pi constant number := 3.141592654;
    -- other declarations
    radius number(5,2);
    dia number(5,2);
    circumference number(7, 2);
    area number (10, 2);
BEGIN
    -- processing
     radius := 10.5;
     dia := radius * 2;
     circumference := 2.0 * pi * radius;
     area := pi * radius * radius;
     -- output
    dbms_output.put_line('Radius: ' || radius);
    dbms_output.put_line('Diameter: ' || dia);
    dbms_output.put_line('Circumference: ' || circumference);
    dbms_output.put_line('Area: ' || area);
END;
 /
```

Output:

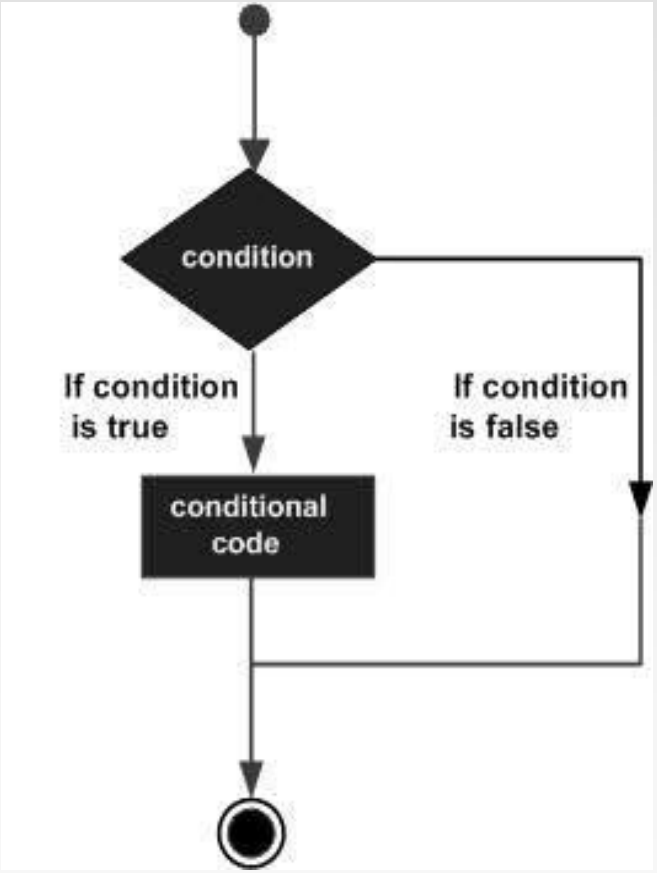
```
Radius: 10.5
Diameter: 21
Circumference: 65.97
Area: 346.36
```

PL/SQL Literals:

- Literals is an explicit numeric, character, string or Boolean values which are not represented by identifiers i.e. TRUE, NULL, etc.
- Note: PL/SQL literals are case-sensitive.
- Types of literals in PL/SQL:
- 1. Numeric Literals (765, 23.56 etc.).
- 2. Character Literals (‘A’ ‘%’ ‘9’ ‘ ‘ ‘z’ etc.).
- 3. String Literals (tutorialspointexamples.com etc.).
- 4. BOOLEAN Literals (TRUE, FALSE and NULL).
- 5. Date and Time Literals (‘2016-12-25’ ‘2016-02-03 12:10:01’ etc.).

PL/SQL - Conditions

- Decision-making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.
- Following is the general form of a typical conditional (i.e., decision making) structure found in most of the programming languages −



PL/SQL - Conditions

- IF - THEN statement
- It is the simplest form of the IF control statement, frequently used in decision-making and changing the control flow of the program execution.
- The IF statement associates a condition with a sequence of statements enclosed by the keywords THEN and END IF. If the condition is TRUE, the statements get executed, and if the condition is FALSE or NULL, then the IF statement does nothing.
- Syntax for IF-THEN statement is −

```
IF condition THEN
  S;
END IF;
```

- Where condition is a Boolean or relational condition and S is a simple or compound statement. Following is an example of the IF-THEN statement −

- IF (a <= 20) THEN
- c:= c+1;
- END IF;
- If the Boolean expression condition evaluates to true, then the block of code inside the if statement will be executed. If the Boolean expression evaluates to false, then the first set of code after the end of the if statement (after the closing end if) will be executed.

Example

```
DECLARE
   a number(2) := 10;
BEGIN
   a:= 10;
  -- check the boolean condition using if statement
   IF( a < 20 ) THEN
      -- if condition is true then print the following
      dbms_output.put_line('a is less than 20 ' );
   END IF;
   dbms_output.put_line('value of a is : ' || a);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
a is less than 20
value of a is : 10

PL/SQL procedure successfully completed.
```

Consider we have a table and few records in the table as we had created in PL/SQL Variable Types

```
DECLARE
   c_id customers.id%type := 1;
   c_sal  customers.salary%type;
BEGIN
   SELECT  salary  INTO  c_sal  FROM customers  WHERE id = c_id;
   IF (c_sal <= 2000) THEN
      UPDATE customers  SET salary =  salary + 1000  WHERE id = c_id;
      dbms_output.put_line ('Salary updated');
   END IF;
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Salary updated

PL/SQL procedure successfully completed.
```

IF-THEN-ELSE statement

- IF statement adds the keyword ELSE followed by an alternative sequence of statement. If the condition is false or NULL, then only the alternative sequence of statements get executed. It ensures that either of the sequence of statements is executed.
- A sequence of IF-THEN statements can be followed by an optional sequence of ELSE statements, which execute when the condition is FALSE.
- Syntax for the IF-THEN-ELSE statement is −

- IF condition THEN
- S1;
- ELSE
- S2;
- END IF;

- Where, S1 and S2 are different sequence of statements. In the IF-THEN-ELSE statements, when the test condition is TRUE, the statement S1 is executed and S2 is skipped; when the test condition is FALSE, then S1 is bypassed and statement S2 is executed.
- For example −

- IF color = red THEN
- dbms_output.put_line('You have chosen a red car')
- ELSE
- dbms_output.put_line('Please choose a color for your car');
- END IF;
- If the Boolean expression condition evaluates to true, then the if-then block of code will be executed otherwise the else block of code will be executed.

Example

- Let us try an example that will help you understand the concept −

```
DECLARE
   a number(3) := 100;
BEGIN
   -- check the boolean condition using if statement
   IF( a < 20 ) THEN
      -- if condition is true then print the following
      dbms_output.put_line('a is less than 20 ' );
   ELSE
      dbms_output.put_line('a is not less than 20 ' );
   END IF;
   dbms_output.put_line('value of a is : ' || a);
END;
/
```

- When the above code is executed at the SQL prompt, it produces the following result −

```
a is not less than 20
value of a is : 100

PL/SQL procedure successfully completed.
```

IF-THEN-ELSIF statement

- The IF-THEN-ELSIF statement allows you to choose between several alternatives. An IF-THEN statement can be followed by an optional ELSIF...ELSE statement. The ELSIF clause lets you add additional conditions.
- When using IF-THEN-ELSIF statements there are a few points to keep in mind.
- It's ELSIF, not ELSEIF.
- An IF-THEN statement can have zero or one ELSE's and it must come after any ELSIF's.
- An IF-THEN statement can have zero to many ELSIF's and they must come before the ELSE.
- Once an ELSIF succeeds, none of the remaining ELSIF's or ELSE's will be tested.

- The syntax of an IF-THEN-ELSIF Statement in PL/SQL programming language is −

- IF(boolean_expression 1)THEN
- S1; -- Executes when the boolean expression 1 is true
- ELSIF(boolean_expression 2) THEN
- S2; -- Executes when the boolean expression 2 is true
- ELSIF(boolean_expression 3) THEN
- S3; -- Executes when the boolean expression 3 is true
- ELSE
- S4; -- executes when the none of the above condition is true
- END IF;

Example

```
DECLARE
   a number(3) := 100;
BEGIN
   IF ( a = 10 ) THEN
      dbms_output.put_line('Value of a is 10' );
   ELSIF ( a = 20 ) THEN
      dbms_output.put_line('Value of a is 20' );
   ELSIF ( a = 30 ) THEN
      dbms_output.put_line('Value of a is 30' );
   ELSE
       dbms_output.put_line('None of the values is matching');
   END IF;
   dbms_output.put_line('Exact value of a is: '|| a );
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

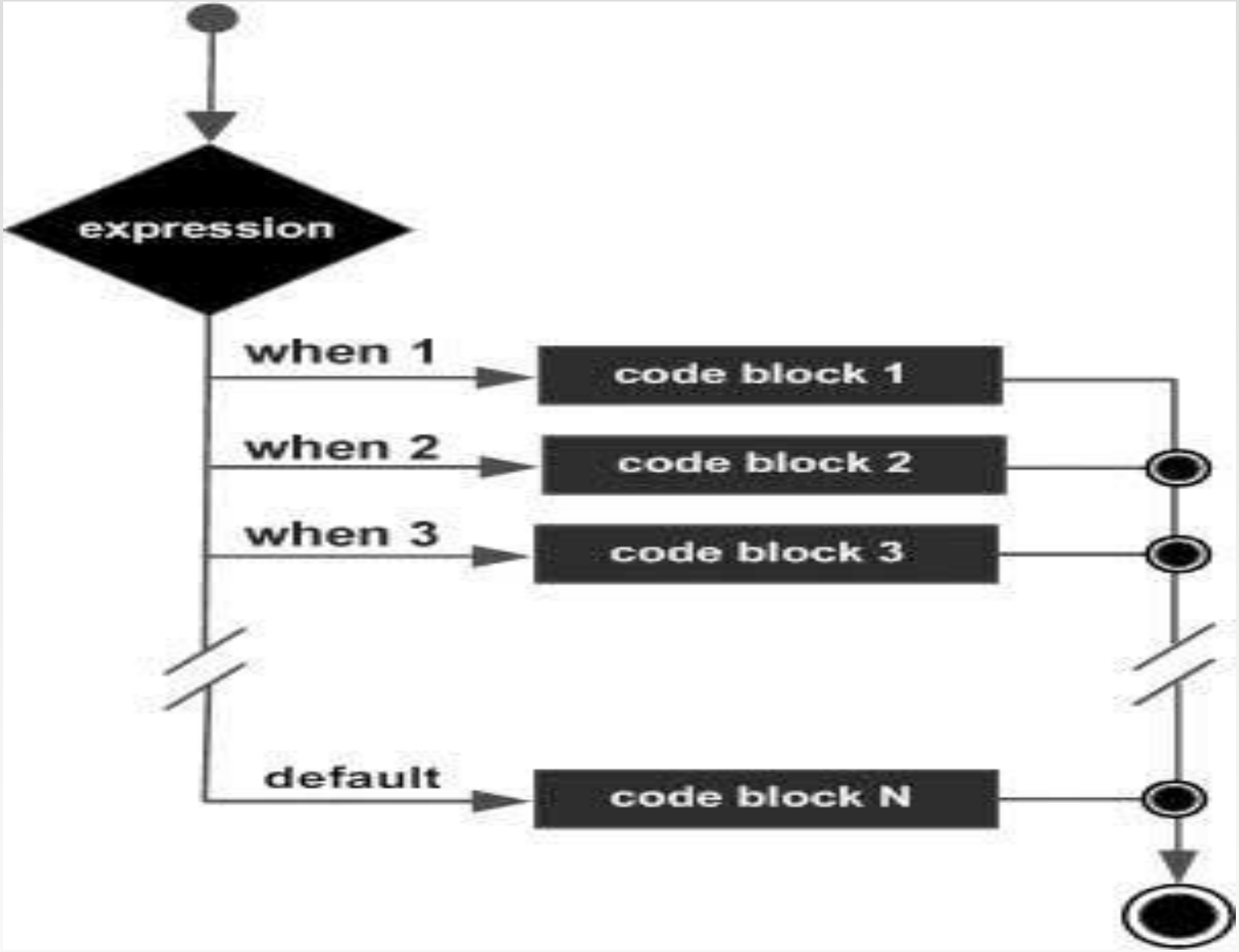
```
None of the values is matching
Exact value of a is: 100


PL/SQL procedure successfully completed.
```

CASE statement

- Like the IF statement, the CASE statement selects one sequence of statements to execute. However, to select the sequence, the CASE statement uses a selector rather than multiple Boolean expressions. A selector is an expression, the value of which is used to select one of several alternatives.
- The syntax for the case statement in PL/SQL is −
- CASE selector
- WHEN 'value1' THEN S1;
- WHEN 'value2' THEN S2;
- WHEN 'value3' THEN S3;
- ...
- ELSE Sn; -- default case
- END CASE;

Flow Diagram



CASE Example

```
DECLARE
   grade char(1) := 'A';
BEGIN
   CASE grade
      when 'A' then dbms_output.put_line('Excellent');
      when 'B' then dbms_output.put_line('Very good');
      when 'C' then dbms_output.put_line('Well done');
      when 'D' then dbms_output.put_line('You passed');
      when 'F' then dbms_output.put_line('Better try again');
      else dbms_output.put_line('No such grade');
   END CASE;
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Excellent

PL/SQL procedure successfully completed.
```

PL/SQL - Searched CASE Statement

- The searched CASE statement has no selector and the WHEN clauses of the statement contain search conditions that give Boolean values.
- The syntax for the searched case statement in PL/SQL is −
- CASE
- WHEN selector = 'value1' THEN S1;
- WHEN selector = 'value2' THEN S2;
- WHEN selector = 'value3' THEN S3;
- ...
- ELSE Sn; -- default case
- END CASE;

Example

```
DECLARE
   grade char(1) := 'B';
BEGIN
   case
      when grade = 'A' then dbms_output.put_line('Excellent');
      when grade = 'B' then dbms_output.put_line('Very good');
      when grade = 'C' then dbms_output.put_line('Well done');
      when grade = 'D' then dbms_output.put_line('You passed');
      when grade = 'F' then dbms_output.put_line('Better try again');
      else dbms_output.put_line('No such grade');
   end case;
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Very good
PL/SQL procedure successfully completed.
```

PL/SQL - Nested IF-THEN-ELSE Statements

- It is always legal in PL/SQL programming to nest the IF-ELSE statements, which means you can use one IF or ELSE IF statement inside another IF or ELSE IF statement(s).

- Syntax
- IF(boolean_expression 1)THEN
- -- executes when the boolean expression 1 is true
- IF(boolean_expression 2) THEN
- -- executes when the boolean expression 2 is true
- sequence-of-statements;
- END IF;
- ELSE
- -- executes when the boolean expression 1 is not true
- else-statements;
- END IF;

Example

```
DECLARE
   a number(3) := 100;
   b number(3) := 200;
BEGIN
   -- check the boolean condition
   IF( a = 100 ) THEN
   -- if condition is true then check the following
      IF( b = 200 ) THEN
      -- if condition is true then print the following
      dbms_output.put_line('Value of a is 100 and b is 200' );
      END IF;
   END IF;
   dbms_output.put_line('Exact value of a is : ' || a );
   dbms_output.put_line('Exact value of b is : ' || b );
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Value of a is 100 and b is 200
Exact value of a is : 100
Exact value of b is : 200
```

Loop statement

- A loop statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages −
- **Basic loop structure** encloses sequence of statements in between the LOOP and END LOOP statements. With each iteration, the sequence of statements is executed and then control resumes at the top of the loop.
- The syntax of a basic loop in PL/SQL programming language is −
- LOOP
- Sequence of statements;
- END LOOP;
- Here, the sequence of statement(s) may be a single statement or a block of statements. An EXIT statement or an EXIT WHEN statement is required to break the loop.

Example

```
DECLARE
   x number := 10;
BEGIN
   LOOP
      dbms_output.put_line(x);
      x := x + 10;
      IF x > 50 THEN
         exit;
      END IF;
   END LOOP;
   -- after exit, control resumes here
   dbms_output.put_line('After Exit x is: ' || x);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
10
20
30
40
50
After Exit x is: 60

PL/SQL procedure successfully completed.
```

You can use the EXIT WHEN statement instead of the EXIT statement −

```
DECLARE
   x number := 10;
BEGIN
   LOOP
      dbms_output.put_line(x);
      x := x + 10;
      exit WHEN x > 50;
   END LOOP;
   -- after exit, control resumes here
   dbms_output.put_line('After Exit x is: ' || x);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
10
20
30
40
50
After Exit x is: 60

PL/SQL procedure successfully completed.
```

WHILE LOOP

- A WHILE LOOP statement in PL/SQL programming language repeatedly executes a target statement as long as a given condition is true.

Syntax

```
WHILE condition LOOP
   sequence_of_statements
END LOOP;
```

Example

```
DECLARE
   a number(2) := 10;
BEGIN
   WHILE a < 20 LOOP
      dbms_output.put_line('value of a: ' || a);
      a := a + 1;
   END LOOP;
END;
/
```

WHILE LOOP Example

- When the above code is executed at the SQL prompt, it produces the following result −

- value of a: 10
- value of a: 11
- value of a: 12
- value of a: 13
- value of a: 14
- value of a: 15
- value of a: 16
- value of a: 17
- value of a: 18
- value of a: 19

- PL/SQL procedure successfully completed.

FOR LOOP

- A FOR LOOP is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.
- **Syntax**

```
FOR counter IN initial_value .. final_value LOOP
    sequence_of_statements;
END LOOP;
```

Following is the flow of control in a For Loop −

- The initial step is executed first, and only once. This step allows you to declare and initialize any loop control variables.
- Next, the condition, i.e., initial_value .. final_value is evaluated. If it is TRUE, the body of the loop is executed. If it is FALSE, the body of the loop does not execute and the flow of control jumps to the next statement just after the for loop.

FOR LOOP

- After the body of the for loop executes, the value of the counter variable is increased or decreased.
- The condition is now evaluated again. If it is TRUE, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes FALSE, the FOR-LOOP terminates.
- Following are some special characteristics of PL/SQL for loop −
- The initial_value and final_value of the loop variable or counter can be literals, variables, or expressions but must evaluate to numbers. Otherwise, PL/SQL raises the predefined exception VALUE_ERROR.
- The initial_value need not be 1; however, the loop counter increment (or decrement) must be 1.

FOR LOOP EXAMPLE

- Example
- DECLARE
- a number(2);
- BEGIN
- FOR a in 10 .. 20 LOOP
- dbms_output.put_line('value of a: ' || a);
- END LOOP;
- END;
- /

- When the above code is executed at the SQL prompt, it produces the following result −

- value of a: 10
- value of a: 11
- value of a: 12
- value of a: 13
- value of a: 14
- value of a: 15
- value of a: 16
- value of a: 17
- value of a: 18
- value of a: 19
- value of a: 20

- PL/SQL procedure successfully completed.

Reverse FOR LOOP Statement

- By default, iteration proceeds from the initial value to the final value, generally upward from the lower bound to the higher bound. You can reverse this order by using the REVERSE keyword. In such case, iteration proceeds the other way. After each iteration, the loop counter is decremented.
- However, you must write the range bounds in ascending (not descending) order. The following program illustrates this −

```
DECLARE
   a number(2) ;
BEGIN
   FOR a IN REVERSE 10 .. 20 LOOP
      dbms_output.put_line('value of a: ' || a);
   END LOOP;
END;
/
```

- When the above code is executed at the SQL prompt, it produces the following result −

- value of a: 20
- value of a: 19
- value of a: 18
- value of a: 17
- value of a: 16
- value of a: 15
- value of a: 14
- value of a: 13
- value of a: 12
- value of a: 11
- value of a: 10
- PL/SQL procedure successfully completed.

- PL/SQL allows using one loop inside another loop. Following section shows a few examples to illustrate the concept.

- The syntax for a nested basic LOOP statement in PL/SQL is as follows −

- LOOP
- Sequence of statements1
- LOOP
- Sequence of statements2
- END LOOP;
- END LOOP;

The syntax for a nested FOR LOOP statement in PL/SQL is as follows

- FOR counter1 IN initial_value1 .. final_value1 LOOP
- sequence_of_statements1
- FOR counter2 IN initial_value2 .. final_value2 LOOP
- sequence_of_statements2
- END LOOP;
- END LOOP;

The syntax for a nested WHILE LOOP statement in PL/SQL is as follows –

- WHILE condition1 LOOP
- sequence_of_statements1
- WHILE condition2 LOOP
- sequence_of_statements2
- END LOOP;
- END LOOP;

Example

The following program uses a nested basic loop to find the prime numbers from 2 to 100 −

```
DECLARE
   i number(3);
   j number(3);
BEGIN
   i := 2;
   LOOP
      j:= 2;
      LOOP
         exit WHEN ((mod(i, j) = 0) or (j = i));
         j := j +1;
      END LOOP;
   IF (j = i ) THEN
      dbms_output.put_line(i || ' is prime');
   END IF;
   i := i + 1;
   exit WHEN i = 50;
   END LOOP;
END;
/
```

- When the above code is executed at the SQL prompt, it produces the following result −
- 2 is prime
- 3 is prime
- 5 is prime
- 7 is prime
- 11 is prime
- 13 is prime
- 17 is prime
- 19 is prime
- 23 is prime
- 29 is prime
- 31 is prime
- 37 is prime
- 41 is prime
- 43 is prime
- 47 is prime
- PL/SQL procedure successfully completed.

Labels

- A PL/SQL label is a way to name a particular part of your program. Syntactically, a label has the format:

  <<identifier>>

- where identifier is a valid PL/SQL identifier (up to 30 characters in length and starting with a letter, as discussed earlier in the section Identifiers). There is no terminator; labels appear directly in front of the thing they're labeling, which must be an executable statement—even if it is merely the NULL statement.
- BEGIN
- ...
- <<the_spot>>
- NULL;

- Because anonymous blocks are themselves executable statements, a label can “name” an anonymous block for the duration of its execution. For example:
- <<insert_but_ignore_dups>>
- BEGIN
- INSERT INTO catalog
- VALUES (...);
- EXCEPTION
- WHEN DUP_VAL_ON_INDEX
- THEN
- NULL;
- END insert_but_ignore_dups;
- One reason you might label a block is to improve the readability of your code. When you give something a name, you self-document that code. You also clarify your own thinking about what that code is supposed to do, sometimes ferreting out errors in the process.

- Another reason to use a block label is to allow you to qualify references to elements from an enclosing block that have duplicate names in the current, nested block. Here's a schematic example:
- <<outerblock>>
- DECLARE
- counter INTEGER := 0;
- BEGIN
- ...
- DECLARE
- counter INTEGER := 1;
- BEGIN
- IF counter = outerblock.counter
- THEN
- ...
- END IF;
- END;
- END;
- Without the block label, there would be no way to distinguish between the two "counter" variables. Again, though, a better solution would probably ...

Labeling a PL/SQL Loop

- PL/SQL loops can be labeled. The label should be enclosed by double angle brackets (<< and >>) and appear at the beginning of the LOOP statement. The label name can also appear at the end of the LOOP statement. You may use the label in the EXIT statement to exit from the loop.
- The following program illustrates the concept −

```
DECLARE
   i number(1);
   j number(1);
BEGIN
   << outer_loop >>
   FOR i IN 1..3 LOOP
      << inner_loop >>
      FOR j IN 1..3 LOOP
         dbms_output.put_line('i is: '|| i || ' and j is: ' || j);
      END loop inner_loop;
   END loop outer_loop;
END;
```

- When the above code is executed at the SQL prompt, it produces the following result −

- i is: 1 and j is: 1
- i is: 1 and j is: 2
- i is: 1 and j is: 3
- i is: 2 and j is: 1
- i is: 2 and j is: 2
- i is: 2 and j is: 3
- i is: 3 and j is: 1
- i is: 3 and j is: 2
- i is: 3 and j is: 3

- PL/SQL procedure successfully completed.

- The Loop Control Statements
- Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

- PL/SQL supports the following control statements. Labeling loops also help in taking the control outside a loop. Click the following links to check their details.

EXIT statement

- The EXIT statement in PL/SQL programming language has the following two usages −

- When the EXIT statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.

- If you are using nested loops (i.e., one loop inside another loop), the EXIT statement will stop the execution of the innermost loop and start executing the next line of code after the block.

- The syntax for an EXIT statement in PL/SQL is as follows −

```
EXIT;
```

```
DECLARE
   a number(2) := 10;
BEGIN
   -- while loop execution
   WHILE a < 20 LOOP
      dbms_output.put_line ('value of a: ' || a);
      a := a + 1;
      IF a > 15 THEN
         -- terminate the loop using the exit statement
         EXIT;
      END IF;
   END LOOP;
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15

PL/SQL procedure successfully completed.
```

The EXIT WHEN Statement

- The EXIT-WHEN statement allows the condition in the WHEN clause to be evaluated. If the condition is true, the loop completes and control passes to the statement immediately after the END LOOP.
- Following are the two important aspects for the EXIT WHEN statement −
- Until the condition is true, the EXIT-WHEN statement acts like a NULL statement, except for evaluating the condition, and does not terminate the loop.
- A statement inside the loop must change the value of the condition.
- The syntax for an EXIT WHEN statement in PL/SQL is as follows −

```
EXIT WHEN condition;
```

- The EXIT WHEN statement replaces a conditional statement like if-then used with the EXIT statement.

- Example
- DECLARE
- a number(2) := 10;
- BEGIN
- -- while loop execution
- WHILE a < 20 LOOP
- dbms_output.put_line ('value of a: ' || a);
- a := a + 1;
- -- terminate the loop using the exit when statement
- EXIT WHEN a > 15;
- END LOOP;
- END;
- /
- When the above code is executed at the SQL prompt, it produces the following result −

- value of a: 10
- value of a: 11
- value of a: 12
- value of a: 13
- value of a: 14
- value of a: 15

- PL/SQL procedure successfully completed.

CONTINUE statement

- The CONTINUE statement causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating. In other words, it forces the next iteration of the loop to take place, skipping any code in between.

- Syntax
- The syntax for a CONTINUE statement is as follows −

- CONTINUE;

- Example
- DECLARE
- a number(2) := 10;
- BEGIN
- -- while loop execution
- WHILE a < 20 LOOP
- dbms_output.put_line ('value of a: ' || a);
- a := a + 1;
- IF a = 15 THEN
- -- skip the loop using the CONTINUE statement
- a := a + 1;
- CONTINUE;
- END IF;
- END LOOP;
- END;
- /
- When the above code is executed at the SQL prompt, it produces the following result −

- value of a: 10
- value of a: 11
- value of a: 12
- value of a: 13
- value of a: 14
- value of a: 16
- value of a: 17
- value of a: 18
- value of a: 19

GOTO statement

- A GOTO statement in PL/SQL programming language provides an unconditional jump from the GOTO to a labeled statement in the same subprogram.

- NOTE − The use of GOTO statement is not recommended in any programming language because it makes it difficult to trace the control flow of a program, making the program hard to understand and hard to modify. Any program that uses a GOTO can be rewritten so that it doesn't need the GOTO.
- Syntax
- The syntax for a GOTO statement in PL/SQL is as follows −
- GOTO label;
- ..
- ..
- << label >>
- statement;

Procedures in PL/SQL

- **A subprogram** is a program unit/module that performs a particular task. These subprograms are combined to form larger programs. This is basically called the 'Modular design'. A subprogram can be invoked by another subprogram or program which is called the **calling program.**
- A subprogram can be created −

At the schema level

Inside a package

Inside a PL/SQL block

- At the schema level, subprogram is a **standalone subprogram.** It is created with the CREATE PROCEDURE or the CREATE FUNCTION statement. It is stored in the database and can be deleted with the DROP PROCEDURE or DROP FUNCTION statement.
- A subprogram created inside a package is a packaged subprogram. It is stored in the database and can be deleted only when the package is deleted with the DROP PACKAGE statement. We will discuss packages in the chapter 'PL/SQL - Packages'.

- PL/SQL subprograms are named PL/SQL blocks that can be invoked with a set of parameters. PL/SQL provides two kinds of subprograms:

Functions − These subprograms return a single value; mainly used to compute and return a value.

Procedures − These subprograms do not return a value directly; mainly used to perform an action.

Parts of a PL/SQL Subprogram

Each PL/SQL subprogram has a name, and may also have a parameter list. Like anonymous PL/SQL blocks, the named blocks will also have the following three parts −

1.Declarative Part

It is an optional part. However, the declarative part for a subprogram does not start with the DECLARE keyword. It contains declarations of types, cursors, constants, variables, exceptions, and nested subprograms. These items are local to the subprogram and cease to exist when the subprogram completes execution.

2.Executable Part

This is a mandatory part and contains statements that perform the designated action.

3.Exception-handling

This is again an optional part. It contains the code that handles run-time errors.

Creating a Procedure

A procedure is created with the CREATE OR REPLACE PROCEDURE statement. The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows −

```
CREATE [OR REPLACE] PROCEDURE procedure_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
{IS | AS}
BEGIN
  < procedure_body >
END procedure_name;
```

- Where procedure-name specifies the name of the procedure.
- [OR REPLACE] option allows the modification of an existing procedure.
- The optional parameter list contains name, mode and types of the parameters. IN represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.
- procedure-body contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalone procedure.

Example

- The following example creates a simple procedure that displays the string 'Hello World!' on the screen when executed.

- CREATE OR REPLACE PROCEDURE greetings
- AS
- BEGIN
- dbms_output.put_line('Hello World!');
- END;
- /
- When the above code is executed using the SQL prompt, it will produce the following result −

- Procedure created.

- Executing a Standalone Procedure
- A standalone procedure can be called in two ways −
- Using the EXECUTE keyword
- Calling the name of the procedure from a PL/SQL block

- The above procedure named 'greetings' can be called with the EXECUTE keyword as −

EXECUTE greetings;

The above call will display −

Hello World

PL/SQL procedure successfully completed.

- The procedure can also be called from another PL/SQL block −

- BEGIN
- greetings;
- END;
- /

The above call will display −

- Hello World

- PL/SQL procedure successfully completed.

- Deleting a Standalone Procedure
- A standalone procedure is deleted with the DROP PROCEDURE statement. Syntax for deleting a procedure is −

- DROP PROCEDURE procedure-name;
- You can drop the greetings procedure by using the following statement −

- DROP PROCEDURE greetings;

Parameter Modes in PL/SQL Subprograms

- The following table lists out the parameter modes in PL/SQL subprograms −

Parameter Mode:

IN:An IN parameter lets you pass a value to the subprogram. It is a read-only parameter. Inside the subprogram, an IN parameter acts like a constant. It cannot be assigned a value. You can pass a constant, literal, initialized variable, or expression as an IN parameter. You can also initialize it to a default value; however, in that case, it is omitted from the subprogram call. It is the default mode of parameter passing. **Parameters are passed by reference.**

OUT:An OUT parameter returns a value to the calling program. Inside the subprogram, an OUT parameter acts like a variable. You can change its value and reference the value after assigning it. **The actual parameter must be variable and it is passed by value.**

IN OUT: An IN OUT parameter passes an initial value to a subprogram and returns an updated value to the caller. It can be assigned a value and the value can be read.**The actual parameter corresponding to an IN OUT formal parameter must be a variable, not a constant or an expression. Formal parameter must be assigned a value. Actual parameter is passed by value.**

IN & OUT Mode Example 1

This program finds the minimum of two values. Here, the procedure takes two numbers using the IN mode and returns their minimum using the OUT parameters.

```
DECLARE
   a number;
   b number;
   c number;
PROCEDURE findMin(x IN number, y IN number, z OUT number) IS
BEGIN
   IF x < y THEN
      z:= x;
   ELSE
      z:= y;
   END IF;
END;
BEGIN
   a:= 23;
   b:= 45;
   findMin(a, b, c);
   dbms_output.put_line(' Minimum of (23, 45) : ' || c);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Minimum of (23, 45) : 23
PL/SQL procedure successfully completed.
```

IN & OUT Mode Example 2

- This procedure computes the square of value of a passed value. This example shows how we can use the same parameter to accept a value and then return another result.

```
DECLARE
   a number;
PROCEDURE squareNum(x IN OUT number) IS
BEGIN
  x := x * x;
END;
BEGIN
   a:= 23;
   squareNum(a);
   dbms_output.put_line(' Square of (23): ' || a);
END;
/
```

- When the above code is executed at the SQL prompt, it produces the following result −

```
Square of (23): 529
```

```
PL/SQL procedure successfully completed.
```

Methods for Passing Parameters

Actual parameters can be passed in three ways −

- Positional notation
- Named notation
- Mixed notation

Positional Notation

- In positional notation, you can call the procedure as −

findMin(a, b, c, d);

In positional notation, the first actual parameter is substituted for the first formal parameter; the second actual parameter is substituted for the second formal parameter, and so on. So, a is substituted for x, b is substituted for y, c is substituted for z and d is substituted for m.

- Named Notation
- In named notation, the actual parameter is associated with the formal parameter using the arrow symbol (=>). The procedure call will be like the following −

- findMin(x => a, y => b, z => c, m => d);
- Mixed Notation
- In mixed notation, you can mix both notations in procedure call; however, the positional notation should precede the named notation.

Function

- Creating a Function
- A standalone function is created using the CREATE FUNCTION statement. The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows −

- CREATE [OR REPLACE] FUNCTION function_name
- [(parameter_name [IN | OUT | IN OUT] type [, ...])]
- RETURN return_datatype
- {IS | AS}
- BEGIN
- < function_body >
- END [function_name];

- Where,
- function-name specifies the name of the function.
- [OR REPLACE] option allows the modification of an existing function.
- The optional parameter list contains name, mode and types of the parameters. IN represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.
- The function must contain a return statement.
- The RETURN clause specifies the data type you are going to return from the function.
- function-body contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalone function.

Example

- The following example illustrates how to create and call a standalone function. This function returns the total number of CUSTOMERS in the customers table.
- We will use the CUSTOMERS table, which we had created in the PL/SQL Variables chapter −
- Select * from customers;
- +----+----------+-----+-----------+----------+
- | ID | NAME | AGE | ADDRESS | SALARY |
- +----+----------+-----+-----------+----------+
- | 1 | Ramesh | 32 | Ahmedabad | 2000.00 |
- | 2 | Khilan | 25 | Delhi | 1500.00 |
- | 3 | kaushik | 23 | Kota | 2000.00 |
- | 4 | Chaitali | 25 | Mumbai | 6500.00 |
- | 5 | Hardik | 27 | Bhopal | 8500.00 |
- | 6 | Komal | 22 | MP | 4500.00 |
- +----+----------+-----+-----------+----------+
- CREATE OR REPLACE FUNCTION totalCustomers
- RETURN number IS
- total number(2) := 0;
- BEGIN
- SELECT count(*) into total FROM customers;
- RETURN total;
- END;
- /

When the above code is executed using the SQL prompt, it will produce the following result −
Function created.

Calling a Function

- While creating a function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task. When a program calls a function, the program control is transferred to the called function.

- A called function performs the defined task and when its return statement is executed or when the last end statement is reached, it returns the program control back to the main program.

- To call a function, you simply need to pass the required parameters along with the function name and if the function returns a value, then you can store the returned value.

Following program calls the function totalCustomers from an anonymous block −

```
DECLARE
   c number(2);
BEGIN
   c := totalCustomers();
   dbms_output.put_line('Total no. of Customers: ' || c);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Total no. of Customers: 6

PL/SQL procedure successfully completed.
```

The following example demonstrates Declaring, Defining, and Invoking a Simple PL/SQL Function that computes and returns the maximum of two values.

```
DECLARE
   a number;
   b number;
   c number;
FUNCTION findMax(x IN number, y IN number)
RETURN number
IS
    z number;
BEGIN
   IF x > y THEN
      z:= x;
   ELSE
      Z:= y;
   END IF;
   RETURN z;
END;
BEGIN
   a:= 23;
   b:= 45;
   c := findMax(a, b);
   dbms_output.put_line(' Maximum of (23,45): ' || c);
END;
```

- When the above code is executed at the SQL prompt, it produces the following result −

- Maximum of (23,45): 45

- PL/SQL procedure successfully completed.

PL/SQL Recursive Functions

- We have seen that a program or subprogram may call another subprogram. When a subprogram calls itself, it is referred to as a recursive call and the process is known as recursion.

- To illustrate the concept, let us calculate the factorial of a number. Factorial of a number n is defined as −

- n! = n*(n-1)!
- = n*(n-1)*(n-2)!
- ...
- = n*(n-1)*(n-2)*(n-3)... 1

The following program calculates the factorial of a given number by calling itself recursively −

```
DECLARE
   num number;
   factorial number;

FUNCTION fact(x number)
RETURN number
IS
   f number;
BEGIN
   IF x=0 THEN
      f := 1;
   ELSE
      f := x * fact(x-1);
   END IF;
RETURN f;
END;
BEGIN
   num:= 6;
   factorial := fact(num);
   dbms_output.put_line(' Factorial '|| num || ' is ' || factorial);
END;
```

When the above code is executed at the SQL prompt, it produces the following result −

Factorial 6 is 720

PL/SQL procedure successfully completed.

Cursor

- Oracle creates a memory area, known as the context area, for processing an SQL statement, which contains all the information needed for processing the statement; for example, the number of rows processed, etc.

- A cursor is a pointer to this context area. PL/SQL controls the context area through a cursor. A cursor holds the rows (one or more) returned by a SQL statement. The set of rows the cursor holds is referred to as the active set.

- You can name a cursor so that it could be referred to in a program to fetch and process the rows returned by the SQL statement, one at a time. There are two types of cursors −

- Implicit cursors
- Explicit cursors

Implicit Cursors

- Implicit cursors are automatically created by Oracle whenever an SQL statement is executed, when there is no explicit cursor for the statement. Programmers cannot control the implicit cursors and the information in it.

- Whenever a DML statement (INSERT, UPDATE and DELETE) is issued, an implicit cursor is associated with this statement. For INSERT operations, the cursor holds the data that needs to be inserted. For UPDATE and DELETE operations, the cursor identifies the rows that would be affected.

- In PL/SQL, you can refer to the most recent implicit cursor as the SQL cursor, which always has attributes such as %FOUND, %ISOPEN, %NOTFOUND, and %ROWCOUNT. The SQL cursor has additional attributes, %BULK_ROWCOUNT and %BULK_EXCEPTIONS, designed for use with the FORALL statement.

Attribute & Description

- 1. %FOUND

Returns TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows or a SELECT INTO statement returned one or more rows. Otherwise, it returns FALSE.

- 2. %NOTFOUND

The logical opposite of %FOUND. It returns TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, it returns FALSE.

- 3. %ISOPEN

Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement.

- 4. %ROWCOUNT

Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement.

Any SQL cursor attribute will be accessed as sql%attribute_name

Example

- We will be using the CUSTOMERS table we had created and used in the previous lectures.

- Select * from customers;

- +----+----------+-----+-----------+----------+
- | ID | NAME | AGE | ADDRESS | SALARY |
- +----+----------+-----+-----------+----------+
- | 1 | Ramesh | 32 | Ahmedabad | 2000.00 |
- | 2 | Khilan | 25 | Delhi | 1500.00 |
- | 3 | kaushik | 23 | Kota | 2000.00 |
- | 4 | Chaitali | 25 | Mumbai | 6500.00 |
- | 5 | Hardik | 27 | Bhopal | 8500.00 |
- | 6 | Komal | 22 | MP | 4500.00 |
- +----+----------+-----+-----------+----------+

The following program will update the table and increase the salary of each customer by 500 and use the SQL%ROWCOUNT attribute to determine the number of rows affected −

```
DECLARE
   total_rows number(2);
BEGIN
   UPDATE customers SET salary = salary + 500;
   IF sql%notfound THEN
      dbms_output.put_line('no customers selected');
   ELSIF sql%found THEN
      total_rows := sql%rowcount;
      dbms_output.put_line( total_rows || ' customers selected ');
   END IF;
END;
```

When the above code is executed at the SQL prompt, it produces the following result −

```
6 customers selected
PL/SQL procedure successfully completed.
```

- If you check the records in customers table, you will find that the rows have been updated −

- Select * from customers;

- +----+----------+-----+-----------+----------+
- | ID | NAME | AGE | ADDRESS | SALARY |
- +----+----------+-----+-----------+----------+
- | 1 | Ramesh | 32 | Ahmedabad | 2500.00 |
- | 2 | Khilan | 25 | Delhi | 2000.00 |
- | 3 | kaushik | 23 | Kota | 2500.00 |
- | 4 | Chaitali | 25 | Mumbai | 7000.00 |
- | 5 | Hardik | 27 | Bhopal | 9000.00 |
- | 6 | Komal | 22 | MP | 5000.00 |
- +----+----------+-----+-----------+----------+

Explicit Cursors

- Explicit cursors are programmer-defined cursors for gaining more control over the context area. An explicit cursor should be defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns more than one row.

- The syntax for creating an explicit cursor is −

- CURSOR cursor_name IS select_statement;
- Working with an explicit cursor includes the following steps −

- Declaring the cursor for initializing the memory
- Opening the cursor for allocating the memory
- Fetching the cursor for retrieving the data
- Closing the cursor to release the allocated memory

- Declaring the Cursor

Declaring the cursor defines the cursor with a name and the associated SELECT statement. For example −

```
CURSOR c_customers IS
SELECT id, name, address FROM customers;
```

- Opening the Cursor

Opening the cursor allocates the memory for the cursor and makes it ready for fetching the rows returned by the SQL statement into it. For example, we will open the above defined cursor as follows −

```
OPEN c_customers;
```

- Fetching the Cursor

Fetching the cursor involves accessing one row at a time. For example, we will fetch rows from the above-opened cursor as follows −

```
FETCH c_customers INTO c_id, c_name, c_addr;
```

- Closing the Cursor

Closing the cursor means releasing the allocated memory. For example, we will close the above-opened cursor as follows −

```
CLOSE c_customers;
```

Example

Following is a complete example to illustrate the concepts of explicit cursors

```
DECLARE
  c_id customers.id%type;
  c_name customers.name%type;
  c_addr customers.address%type;
  CURSOR c_customers is
    SELECT id, name, address FROM customers;
BEGIN
  OPEN c_customers;
  LOOP
  FETCH c_customers into c_id, c_name, c_addr;
    EXIT WHEN c_customers%notfound;
    dbms_output.put_line(c_id || ' ' || c_name || ' ' || c_addr);
  END LOOP;
  CLOSE c_customers;
END;
```

- When the above code is executed at the SQL prompt, it produces the following result −

- 1 Ramesh Ahmedabad
- 2 Khilan Delhi
- 3 kaushik Kota
- 4 Chaitali Mumbai
- 5 Hardik Bhopal
- 6 Komal MP
-
- PL/SQL procedure successfully completed.

Triggers

- Triggers are stored programs, which are automatically executed or fired when some events occur. Triggers are, in fact, written to be executed in response to any of the following events −

A database manipulation (DML) statement (DELETE, INSERT, or UPDATE)

A database definition (DDL) statement (CREATE, ALTER, or DROP).

A database operation (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).

- Triggers can be defined on the table, view, schema, or database with which the event is associated.
- **Benefits of Triggers**

- Triggers can be written for the following purposes −
- Generating some derived column values automatically
- Enforcing referential integrity
- Event logging and storing information on table access
- Auditing
- Synchronous replication of tables
- Imposing security authorizations
- Preventing invalid transactions

Creating Triggers

- The syntax for creating a trigger is −

- CREATE [OR REPLACE] TRIGGER trigger_name
- {BEFORE | AFTER | INSTEAD OF }
- {INSERT [OR] | UPDATE [OR] | DELETE}
- [OF col_name]
- ON table_name
- [REFERENCING OLD AS o NEW AS n]
- [FOR EACH ROW]
- WHEN (condition)
- DECLARE
- Declaration-statements
- BEGIN
- Executable-statements
- EXCEPTION
- Exception-handling-statements
- END;

- Where,
- CREATE [OR REPLACE] TRIGGER trigger_name − Creates or replaces an existing trigger with the trigger_name.
- {BEFORE | AFTER | INSTEAD OF} − This specifies when the trigger will be executed. The INSTEAD OF clause is used for creating trigger on a view.
- {INSERT [OR] | UPDATE [OR] | DELETE} − This specifies the DML operation.
- [OF col_name] − This specifies the column name that will be updated.
- [ON table_name] − This specifies the name of the table associated with the trigger.
- [REFERENCING OLD AS o NEW AS n] − This allows you to refer new and old values for various DML statements, such as INSERT, UPDATE, and DELETE.
- [FOR EACH ROW] − This specifies a row-level trigger, i.e., the trigger will be executed for each row being affected. Otherwise the trigger will execute just once when the SQL statement is executed, which is called a table level trigger.
- WHEN (condition) − This provides a condition for rows for which the trigger would fire. This clause is valid only for row-level triggers.

Example

- To start with, we will be using the CUSTOMERS table we had created and used in the previous chapters −
- Select * from customers;

- +----+----------+-----+-----------+----------+
- | ID | NAME | AGE | ADDRESS | SALARY |
- +----+----------+-----+-----------+----------+
- | 1 | Ramesh | 32 | Ahmedabad | 2000.00 |
- | 2 | Khilan | 25 | Delhi | 1500.00 |
- | 3 | kaushik | 23 | Kota | 2000.00 |
- | 4 | Chaitali | 25 | Mumbai | 6500.00 |
- | 5 | Hardik | 27 | Bhopal | 8500.00 |
- | 6 | Komal | 22 | MP | 4500.00 |
- +----+----------+-----+-----------+----------+

The following program creates a row-level trigger for the customers table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table. This trigger will display the salary difference between the old values and new values −

```
CREATE OR REPLACE TRIGGER display_salary_changes
BEFORE DELETE OR INSERT OR UPDATE ON customers
FOR EACH ROW
WHEN (NEW.ID > 0)
DECLARE
   sal_diff number;
BEGIN
   sal_diff := :NEW.salary  - :OLD.salary;
   dbms_output.put_line('Old salary: ' || :OLD.salary);
   dbms_output.put_line('New salary: ' || :NEW.salary);
   dbms_output.put_line('Salary difference: ' || sal_diff);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

Trigger created.

- The following points need to be considered here −

- OLD and NEW references are not available for table-level triggers, rather you can use them for record-level triggers.

- If you want to query the table in the same trigger, then you should use the AFTER keyword, because triggers can query the table or change it again only after the initial changes are applied and the table is back in a consistent state.

- The above trigger has been written in such a way that it will fire before any DELETE or INSERT or UPDATE operation on the table, but you can write your trigger on a single or multiple operations, for example BEFORE DELETE, which will fire whenever a record will be deleted using the DELETE operation on the table.

Triggering a Trigger

- Let us perform some DML operations on the CUSTOMERS table. Here is one INSERT statement, which will create a new record in the table −

- INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
- VALUES (7, 'Kriti', 22, 'HP', 7500.00);
- When a record is created in the CUSTOMERS table, the above create trigger, display_salary_changes will be fired and it will display the following result −

- Old salary:
- New salary: 7500
- Salary difference:

- Because this is a new record, old salary is not available and the above result comes as null. Let us now perform one more DML operation on the CUSTOMERS table. The UPDATE statement will update an existing record in the table −

- UPDATE customers
- SET salary = salary + 500
- WHERE id = 2;
- When a record is updated in the CUSTOMERS table, the above create trigger, display_salary_changes will be fired and it will display the following result −

- Old salary: 1500
- New salary: 2000
- Salary difference: 500

Exception

- An exception is an error condition during a program execution. PL/SQL supports programmers to catch such conditions using EXCEPTION block in the program and an appropriate action is taken against the error condition. There are two types of exceptions −
- System-defined exceptions
- User-defined exceptions

Syntax for Exception Handling

- The general syntax for exception handling is as follows. Here you can list down as many exceptions as you can handle. The default exception will be handled using **WHEN** others **THEN** −
- DECLARE
- <declarations section>
- BEGIN
- <executable command(s)>
- EXCEPTION
- <exception handling goes here >
- WHEN exception1 THEN
- exception1-handling-statements
- WHEN exception2 THEN
- exception2-handling-statements
- WHEN exception3 THEN
- exception3-handling-statements
- .....
- WHEN others THEN
- exception3-handling-statements
- END;

Example

Let us write a code to illustrate the concept. We will be using the CUSTOMERS table we had created and used in the previous lectures −

```
DECLARE
   c_id customers.id%type := 8;
   c_name customerS.Name%type;
   c_addr customers.address%type;
BEGIN
   SELECT  name, address INTO  c_name, c_addr
   FROM customers
   WHERE id = c_id;
   DBMS_OUTPUT.PUT_LINE ('Name: '||  c_name);
   DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);
EXCEPTION
   WHEN no_data_found THEN
      dbms_output.put_line('No such customer!');
   WHEN others THEN
      dbms_output.put_line('Error!');
END;
```

- When the above code is executed at the SQL prompt, it produces the following result −

- No such customer!

- PL/SQL procedure successfully completed.

The above program displays the name and address of a customer whose ID is given. Since there is no customer with ID value 8 in our database, the program raises the run-time exception NO_DATA_FOUND, which is captured in the EXCEPTION block.

Raising Exceptions

- Exceptions are raised by the database server automatically whenever there is any internal database error, but exceptions can be raised explicitly by the programmer by using the command RAISE. Following is the simple syntax for raising an exception −

```
DECLARE
   exception_name EXCEPTION;
BEGIN
   IF condition THEN
      RAISE exception_name;
   END IF;
EXCEPTION
   WHEN exception_name THEN
   statement;
END;
```

You can use the above syntax in raising the Oracle standard exception or any user-defined exception.
In the next section, we will give you an example on raising a user-defined exception. You can raise the Oracle standard exceptions in a similar way.

- User-defined Exceptions
- PL/SQL allows you to define your own exceptions according to the need of your program. A user-defined exception must be declared and then raised explicitly, using either a RAISE statement or the procedure DBMS_STANDARD.RAISE_APPLICATION_ERROR.

- The syntax for declaring an exception is −

- DECLARE
- my-exception EXCEPTION;

Example

The following example illustrates the concept. This program asks for a customer ID, when the user enters an invalid ID, the exception invalid_id is raised.

```
DECLARE
  c_id customers.id%type := &cc_id;
  c_name customerS.Name%type;
  c_addr customers.address%type;
  -- user defined exception
  ex_invalid_id  EXCEPTION;
BEGIN
  IF c_id <= 0 THEN
    RAISE ex_invalid_id;
  ELSE
    SELECT  name, address INTO  c_name, c_addr
    FROM customers
    WHERE id = c_id;
    DBMS_OUTPUT.PUT_LINE ('Name: '||  c_name);
    DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);
  END IF;

EXCEPTION
  WHEN ex_invalid_id THEN
    dbms_output.put_line('ID must be greater than zero!');
  WHEN no_data_found THEN
    dbms_output.put_line('No such customer!');
  WHEN others THEN
    dbms_output.put_line('Error!');
END;
```

Output

- When the above code is executed at the SQL prompt, it produces the following result −

- Enter value for cc_id: -6 (let's enter a value -6)
- old 2: c_id customers.id%type := &cc_id;
- new 2: c_id customers.id%type := -6;
- ID must be greater than zero!
-
- PL/SQL procedure successfully completed.

Pre-defined Exceptions

- PL/SQL provides many pre-defined exceptions, which are executed when any database rule is violated by a program. For example, the predefined exception NO_DATA_FOUND is raised when a SELECT INTO statement returns no rows. The following table lists few of the important pre-defined exceptions −

Pre-defined Exceptions

- ACCESS_INTO_NULL : It is raised when a null object is automatically assigned a value.
- CASE_NOT_FOUND: It is raised when none of the choices in the WHEN clause of a CASE statement is selected, and there is no ELSE clause.
- COLLECTION_IS_NULL: It is raised when a program attempts to apply collection methods other than EXISTS to an uninitialized nested table or varray, or the program attempts to assign values to the elements of an uninitialized nested table or varray.
- DUP_VAL_ON_INDEX : It is raised when duplicate values are attempted to be stored in a column with unique index.
- INVALID_CURSOR: It is raised when attempts are made to make a cursor operation that is not allowed, such as closing an unopened cursor.
- INVALID_NUMBER: It is raised when the conversion of a character string into a number fails because the string does not represent a valid number.

Pre-defined Exceptions

- LOGIN_DENIED: It is raised when a program attempts to log on to the database with an invalid username or password.
- NO_DATA_FOUND: It is raised when a SELECT INTO statement returns no rows.
- NOT_LOGGED_ON: It is raised when a database call is issued without being connected to the database.
- PROGRAM_ERROR: It is raised when PL/SQL has an internal problem.
- ROWTYPE_MISMATCH : It is raised when a cursor fetches value in a variable having incompatible data type.
- SELF_IS_NULL : It is raised when a member method is invoked, but the instance of the object type was not initialized.
- STORAGE_ERROR: It is raised when PL/SQL ran out of memory or memory was corrupted.
- TOO_MANY_ROWS: It is raised when a SELECT INTO statement returns more than one row.
- VALUE_ERROR: It is raised when an arithmetic, conversion, truncation, or sizeconstraint error occurs.
- ZERO_DIVIDE : It is raised when an attempt is made to divide a number by zero.