Concurrency Control Technique

Concurrency Control

- Concurrency control is the procedure in DBMS for managing simultaneous operations without conflicting with each another.

- Concurrency control is used to address such conflicts which mostly occur with a multi-user system. It helps you to make sure that database transactions are performed concurrently without violating the data integrity of respective databases.

- Assume that two people who go to electronic kiosks at the same time to buy a movie ticket for the same movie and the same show time.
- However, there is only one seat left in for the movie show in that particular theatre. Without concurrency control, it is possible that both moviegoers will end up purchasing a ticket. However, concurrency control method does not allow this to happen. Both moviegoers can still access information written in the movie seating database. But concurrency control only provides a ticket to the buyer who has completed the transaction process first.

Concurrency Control Protocols

- Different concurrency control protocols offer different benefits between the amount of concurrency they allow and the amount of overhead that they impose.

- Lock-Based Protocols
- Timestamp-Based Protocols
- Validation-Based Protocols

Lock-based Protocols

LOCK

- A lock is a data variable which is associated with a data item. This lock signifies that operations that can be performed on the data item. Locks help synchronize access to the database items by concurrent transactions.

- All lock requests are made to the concurrency-control manager. Transactions proceed only once the lock request is granted.

- **1. Shared Lock (S):**
- A shared lock is also called a Read-only lock. With the shared lock, the data item can be shared between transactions. This is because you will never have permission to update data on the data item.
- For example, consider a case where two transactions are reading the account balance of a person. The database will let them read by placing a shared lock. However, if another transaction wants to update that account's balance, shared lock prevent it until the reading process is over.

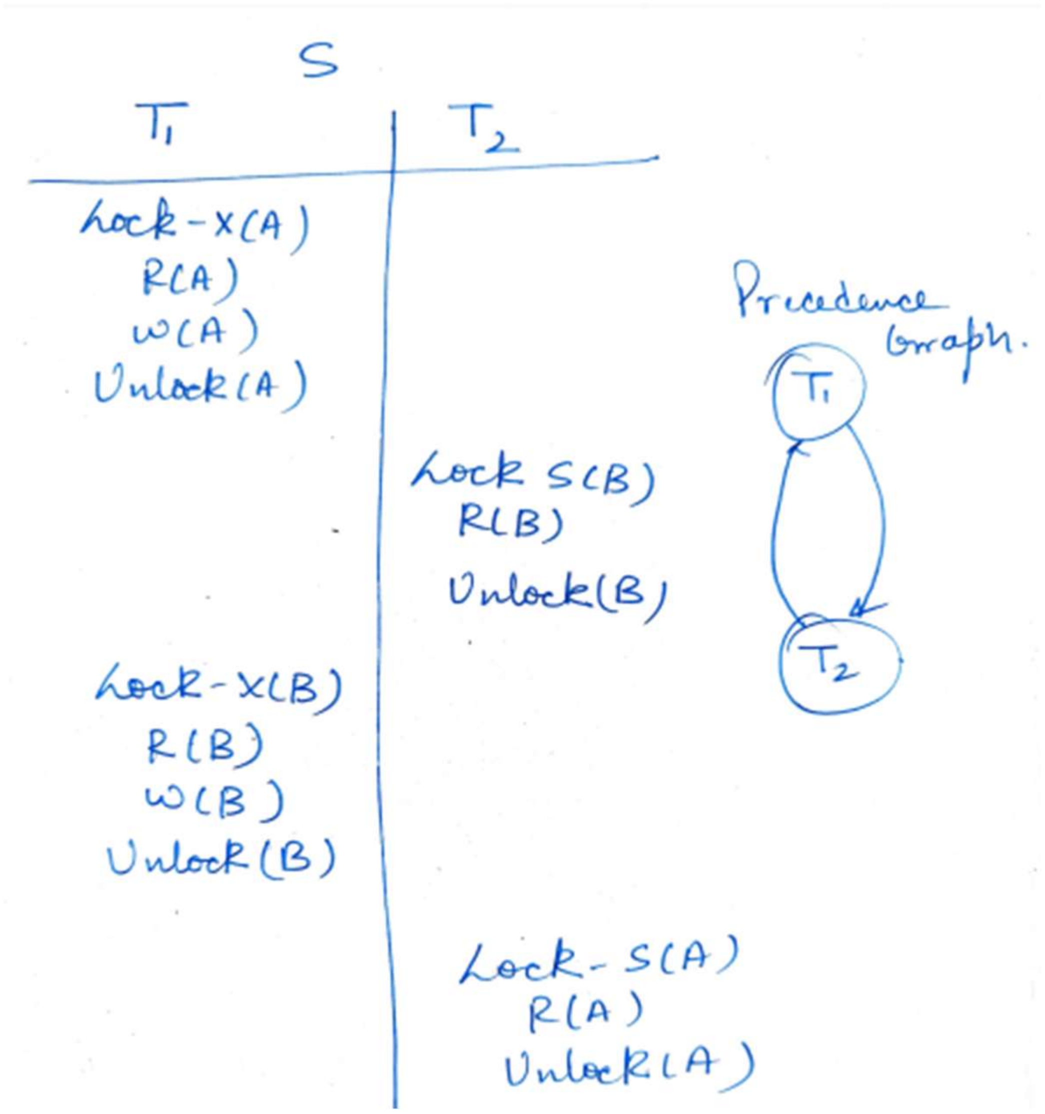
- **2. Exclusive Lock (X):**
- With the Exclusive Lock, a data item can be read as well as written. This is exclusive and can't be held concurrently on the same data item. X-lock is requested using lock-x instruction. Transactions may unlock the data item after finishing the 'write' operation.
- For example, when a transaction needs to update the account balance of a person. You can allows this transaction by placing X lock on it. Therefore, when the second transaction wants to read or write, exclusive lock prevent this operation.

Two Phase Locking (2PL) Protocol

- The Two-Phase Locking protocol allows each transaction to make a lock or unlock request in two steps:
- **Growing Phase**: In this phase transaction may obtain locks but may not release any locks.
- **Shrinking Phase**: In this phase, a transaction may release locks but not obtain any new lock

It is true that the 2PL protocol offers serializability. However, it does not ensure that deadlocks do not happen.

Lock based Protocol



S

T_1	T_2
Lock-X(A)	
R(A)	
W(A)	
Unlock(A)	
	Lock S(B)
	R(B)
	Unlock(B)
Lock-X(B)	
R(B)	
W(B)	
Unlock(B)	
	Lock-S(A)
	R(A)
	UnlockR(A)

Precedence Graph.

$$\begin{tikzcd} T_1 \arrow[d, bend left] \\ T_2 \arrow[u, bend left] \end{tikzcd}$$

Types of 2PL

1. Basic 2PL
2. Conservative 2PL
3. Strict 2PL
4. Rigorous 2PL

Basic 2PL

	T1	T2
0	LOCK-S(A)	
1		LOCK-S(A)
2	LOCK-X(B)	
3	——	——
4	UNLOCK(A)	
5		LOCK-X(C)
6	UNLOCK(B)	
7		UNLOCK(A)
8		UNLOCK(C)
9	——	——

The following way shows how unlocking and locking work with 2-PL.

Transaction T1:

•**Growing phase:** from step 1-3

•**Shrinking phase:** from step 5-7

•**Lock point:** at 3

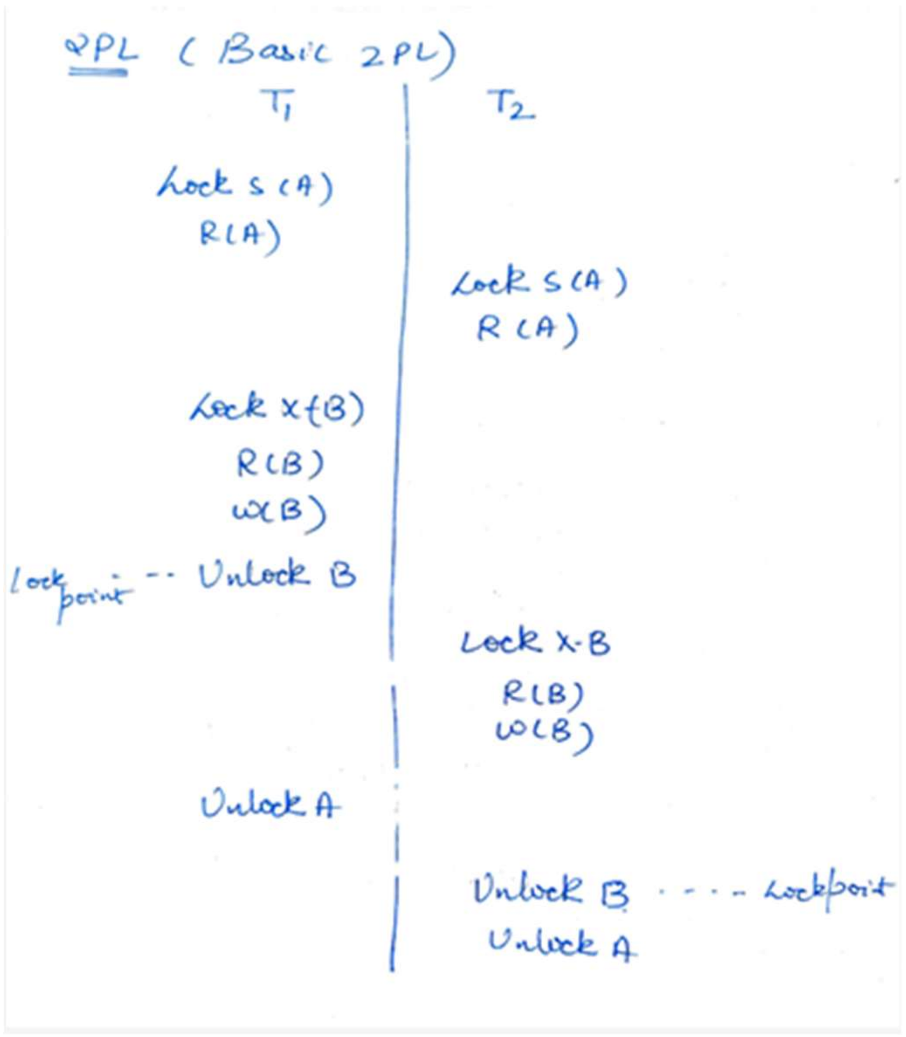
Transaction T2:

•**Growing phase:** from step 2-6

•**Shrinking phase:** from step 8-9

•**Lock point:** at 6

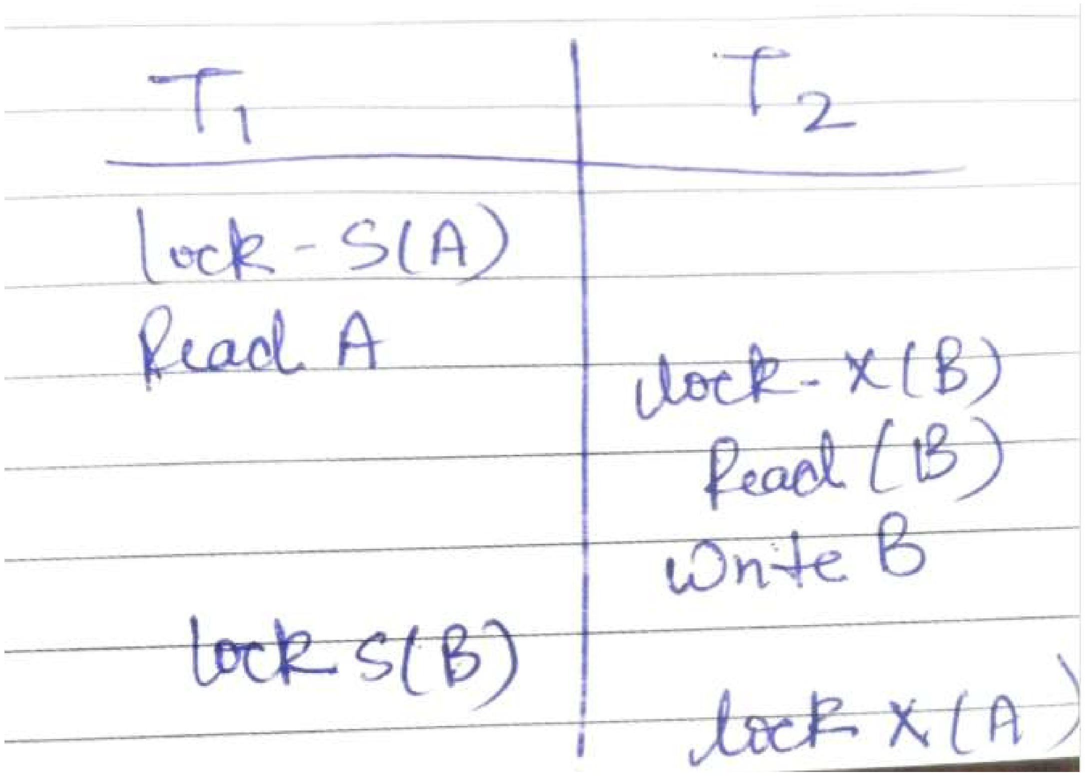
Basic 2 PL Example



2PL (Basic 2PL)

T1	T2
Lock S(A)	
R(A)	
	Lock S(A)
	R(A)
Lock X(B)	
R(B)	
W(B)	
Lock point -- Unlock B	
	Lock X-B
	R(B)
	W(B)
Unlock A	
	Unlock B ---- Lockpoint
	Unlock A

- 1. Ensure conflict and view serializabilty
- 2. Does not ensure freedom from deadlock



T_1	T_2
lock-S(A)	
Read A	
	lock-X(B)
	Read (B)
	Write B
lock-S(B)	
	lock-X(A)

Dealing with Deadlock and Starvation

- **Deadlock**

T’1	**T’2**
read_lock (Y);	
read_item (Y);	
	read_lock (X);
	read_item (Y);
write_lock (X);	
(waits for X)	write_lock (Y);
	(waits for Y)

T1 and T2 did follow two-phase policy but they are deadlock

- Deadlock (T’1 and T’2)

Conservative / Static 2PL

- **Static 2-PL**, this protocol requires the transaction to lock all the items it access before the Transaction begins execution by predeclaring its read-set and write-set. If any of the predeclared items needed cannot be locked, the transaction does not lock any of the items, instead, it waits until all the items are available for locking. So the operation on data cannot start until we lock all the items required.

- Independent from deadlock
- May have irrecoverability and cascading rollback

	T_1	T_2
1	Lock-X(A)	
2	Lock-X(B)	
3	Read(A)	
4	*operation on A	
5	Write(A)	
6	Unlock(A)	
7		Lock-X(A)
8		Read(A)
9		*operation on A
10		Write(A)
11		Unlock(A)
12	Read(B)	
13	*operation on B	
14	Write(B)	
15	Unlock(B)	
16	**Commit**	
17		**Commit**

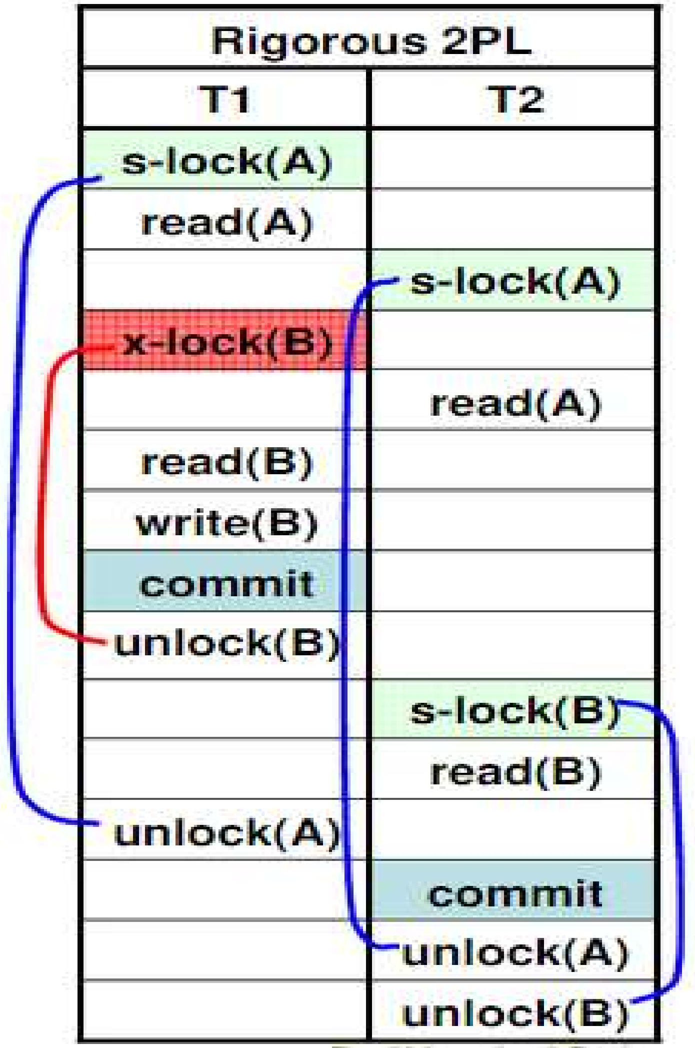
Rigourous 2PL

The two rules of Rigorous 2PL are:

If a transaction T wants to read/write an object, it must request a shared/exclusive lock on the object.

All locks (both exclusive **and shared**) held by transaction T are released when T commits (and not before).

- Under rigorous 2PL, transactions can be serialized by the order in which they commit.



Rigorous 2PL	
T1	T2
s-lock(A)	
read(A)	
	s-lock(A)
x-lock(B)	
	read(A)
read(B)	
write(B)	
commit	
unlock(B)	
	s-lock(B)
	read(B)
unlock(A)	
	commit
	unlock(A)
	unlock(B)

Strict 2PL

- Here a transaction must hold all its exclusive locks till it commits/aborts.

Strict 2PL	
T1	T2
s-lock(A)	
read(A)	
	s-lock(A)
x-lock(B)	
unlock(A)	
read(B)	
write(B)	
	read(A)
	unlock(A)
commit	
unlock(B)	
	s-lock(B)
	read(B)
	unlock(B)
	commit

Timestamp Based Protocol

Timestamp Ordering Protocol –

- The main idea for this protocol is to order the transactions based on their Timestamps.
- Stating simply, the schedule is equivalent to the particular *Serial Order* corresponding to the *order of the Transaction timestamps*.

Timestamp

- A monotonically increasing variable (integer) indicating the age of an operation or a transaction. A larger timestamp value indicates a more recent event or operation.
- Timestamp based algorithm uses timestamp to serialize the execution of concurrent transactions.

- Typically, **timestamp** values are assigned in the order in which the **transactions** are submitted to the system. So, a **timestamp** can be thought of as the **transaction** start time. Therefore, time stamping is a method of concurrency control in which each **transaction** is assigned a **transaction timestamp** →TS(Ti)

- To ensure this, use two Timestamp Values relating to each database item **X**.
- **W_TS(X)** is the latest timestamp of any transaction that executed **write(X)** successfully.
- **R_TS(X)** is the latest timestamp of any transaction that executed **read(X)** successfully.

- **If a transaction Ti issues a read(X) operation −**
    - If TS(Ti) < W-timestamp(X)
        - Operation rejected.
    - If TS(Ti) >= W-timestamp(X)
        - Operation executed.
    - All data-item timestamps updated.
- **If a transaction Ti issues a write(X) operation −**
    - If TS(Ti) < R-timestamp(X)
        - Operation rejected.
    - If TS(Ti) < W-timestamp(X)
        - Operation rejected and Ti rolled back.
    - Otherwise, operation executed.

Properties of timestamp based protocol

- 1. ensure conflict serializabilty
- 2. ensure view serializability
- 3. possibility of dirty read
- 4. no chance of deadlock
- 5. may have starvation

Thomas' Write Rule

This rule states if TS(Ti) < W-timestamp(X), then the operation is rejected and T_i is rolled back.
Time-stamp ordering rules can be modified to make the schedule view serializable.
Instead of making T_i rolled back, the 'write' operation itself is ignored.

Validation based protocol

- **Validation Based Protocol** is also called Optimistic Concurrency Control Technique. This protocol is used in DBMS (Database Management System) for avoiding concurrency in transactions. It is called optimistic because of the assumption it makes, i.e. very less interference occurs therefore there is no need for checking while the transaction is executed.

- In this technique, no checking is done while the transaction is been executed. Until the transaction end is reached updates in the transaction are not applied directly to the database.
- All updates are applied to local copies of data items kept for transaction. At the end of transaction execution, while execution of transaction, a **validation phase** checks whether any of transaction updates violate serializability. If there is no violation of serializability the transaction is committed and the database is updated; or else, the transaction is rolled back and then restarted.

Optimistic Concurrency Control is a three-phase protocol. The three phases for validation-based protocol:

- **Read Phase:**
  Values of committed data items from the database can be read by a transaction. Updates are only applied to local data versions.
- **Validation Phase:**
  Checking is performed to make sure that there is no violation of serializability when the transaction updates are applied to database.
- **Write Phase:**
  On the success of validation phase, the transaction updates are applied to the database, otherwise, the updates are discarded and the transaction is slowed down.

Validation Phase

- 3 Timestamp

1. Start T – Start of Execution of T
2. Validation T – Start of Validation Phase of T
3. Finish T – End of Write Phase of T

Validation Test

1. Write_set: of a transaction contains all the write operations that T_i performs.

2. Read_set: of a transaction contains all the read operations that T_i performs.

In the Validation phase for transaction T_i the protocol inspect that T_i doesn’t overlap or intervene with any other transactions currently in their validation phase or in committed. The validation phase for T_i checks that for all transaction T_j one of the following below conditions must hold to being validated or pass validation phase:

- **1. Finish(T_j)<Starts(T_i)**, since T_j finishes its execution means completes its write-phase before T_i started its execution(read-phase). Then the serializability indeed maintained.
- **2.** T_i begins its write phase after T_j completes its write phase, and the read_set of T_i should be disjoint with write_set of T_j.
- **3.** T_j completes its read phase before T_i completes its read phase and both read_set and write_set of T_i are disjoint with the write_set of T_j.

Multiple Granularity

- **Granularity:** It is the size of data item allowed to lock.

- Multiple Granularity:
- It can be defined as hierarchically breaking up the database into blocks which can be locked.
- The Multiple Granularity protocol enhances concurrency and reduces lock overhead.
- It maintains the track of what to lock and how to lock.
- It makes easy to decide either to lock a data item or to unlock a data item. This type of hierarchy can be graphically represented as a tree.

- Hence, the levels of the tree starting from the top level are as follows:
    1. Database
    2. Area
    3. File
    4. Record

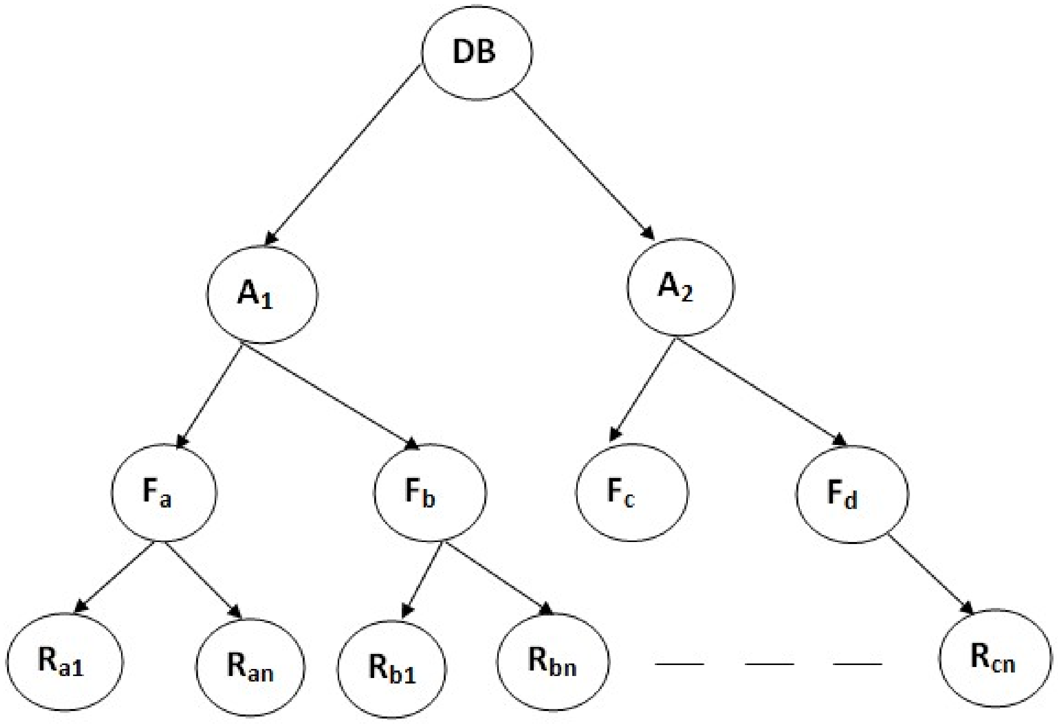


Figure: Multi Granularity tree Hierarchy