



TRANSACTIONS

Introduction to Database Systems

*Mahdi Akhi
Sharif University of Technology*

IN THIS LECTURE

- Transactions
- Recovery
 - System and Media Failures
- Concurrency
 - Concurrency problems
- For more information
 - Connolly and Begg chapter 20
 - Ullman and Widom 8.6

TRANSACTIONS

- A transaction is an action, or a series of actions, carried out by a single user or an application program, which reads or updates the contents of a database.

TRANSACTIONS

- A transaction is a 'logical unit of work' on a database
- Each transaction does something in the database
- No part of it alone achieves anything of use or interest
- Transactions are the unit of recovery, consistency, and integrity as well
- **ACID** properties
 - **A**tomicity
 - **C**onsistency
 - **I**solation
 - **D**urability

ATOMICITY AND CONSISTENCY

➤ Atomicity

- Transactions are atomic
 - they don't have parts (conceptually)
- **Can't be executed partially**; it should not be detectable that they interleave with another transaction

➤ Consistency

- Transactions take the database from one consistent state into another
- In the middle of a transaction the database might not be consistent

ISOLATION AND DURABILITY

➤ Isolation

- The **effects** of a transaction are **not visible** to other transactions until it has completed
- From outside the transaction has either happened or not
- To me this actually sounds like a consequence of atomicity...

➤ Durability

- Once a transaction has completed, its changes are made permanent
- Even if the system crashes, the effects of a transaction must remain in place

EXAMPLE OF TRANSACTION

- Transfer £50 from account A to account B

Read(A)
A = A - 50
Write(A)
Read(B)
B = B + 50
Write(B)

} transaction

- **Atomicity** - shouldn't take money from A without giving it to B
- **Consistency** - money isn't lost or gained
- **Isolation** - other queries shouldn't see A or B change until completion
- **Durability** - the money does not go back to A

THE TRANSACTION MANAGER

- The transaction manager enforces the **ACID** properties
 - It schedules the operations of transactions
 - **COMMIT** and **ROLLBACK** are used to ensure atomicity
- Locks or timestamps are used to ensure consistency and isolation for concurrent transactions
- A log is kept to ensure durability in the event of system failure

COMMIT AND ROLLBACK

- **COMMIT** signals the successful end of a transaction
 - Any changes made by the transaction should be saved
 - These changes are now visible to other transactions
- **ROLLBACK** signals the unsuccessful end of a transaction
 - Any changes made by the transaction should be undone
 - It is now as if the transaction never existed

RECOVERY

- Transactions should be durable, but we cannot prevent all sorts of failures:
 - System crashes
 - Power failures
 - Disk crashes
 - User mistakes
 - Sabotage
 - Natural disasters
- Prevention is better than cure
 - Reliable OS
 - Security
 - UPS and surge protectors
 - RAID arrays
 - Can't protect against everything though

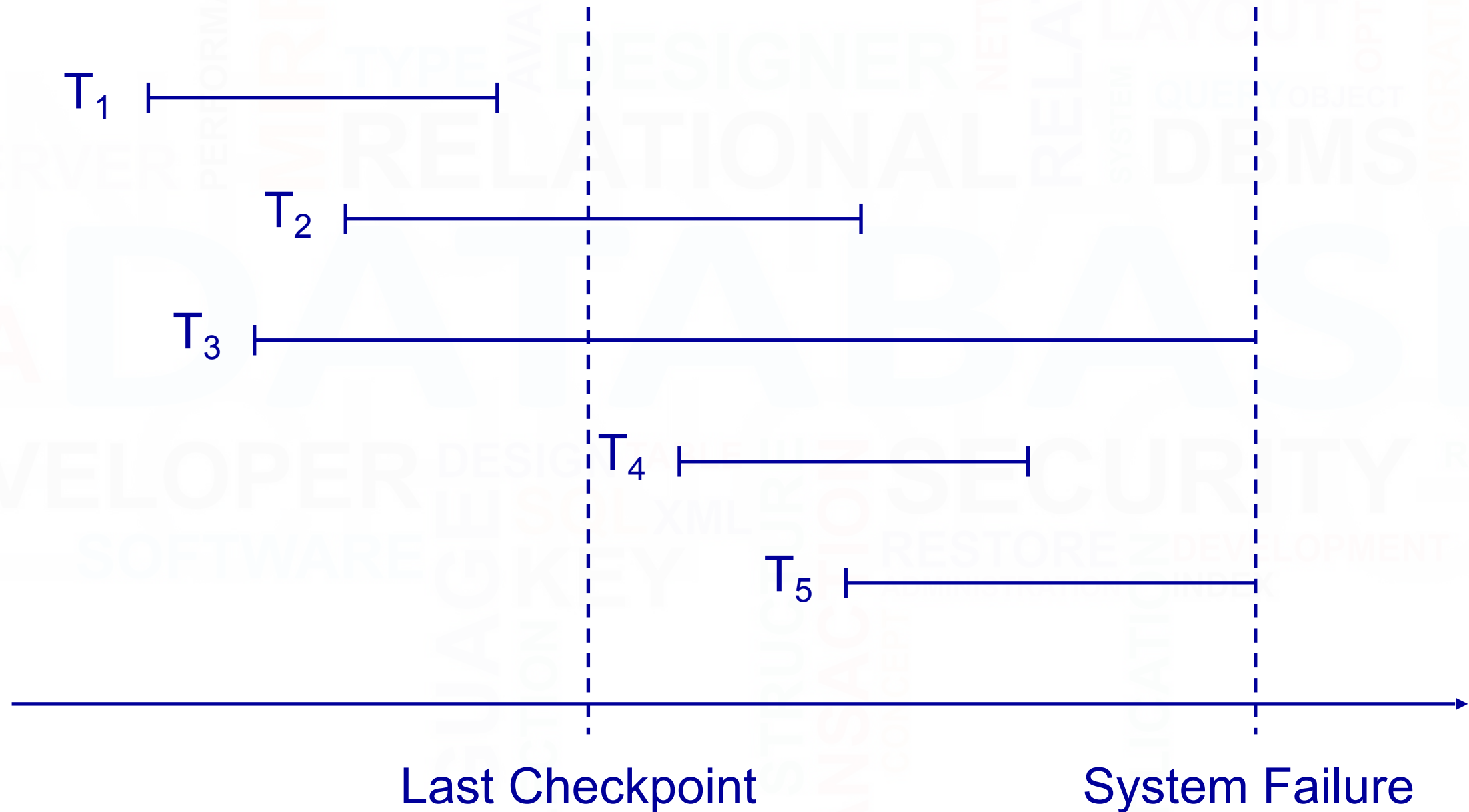
THE TRANSACTION LOG

- The transaction log records the **details of all transactions**
 - Any changes the transaction makes to the database
 - How to undo these changes
 - When transactions complete and how
- The log is stored on disk, not in memory
 - If the system crashes it is preserved
- Write ahead log rule
 - The entry in the log must be made before **COMMIT** processing can complete

SYSTEM FAILURES

- A system failure means all running transactions are affected
 - **Software crashes**
 - **Power failures**
- The physical media (disks) are not damaged
- At various times a DBMS takes a checkpoint
 - All committed transactions are written to disk
 - A record is made (on disk) of the transactions that are currently running

TYPES OF TRANSACTIONS



SYSTEM RECOVERY

- Any transaction that was **running** at the time of failure needs to be **undone** and restarted
- Any transactions that **committed** since the last checkpoint need to be **redone**
- Transactions of type T_1 need no recovery
- Transactions of type T_3 or T_5 need to be undone and restarted
- Transactions of type T_2 or T_4 need to be redone

TRANSACTION RECOVERY

UNDO and REDO: lists of transactions

UNDO = all transactions running at the last checkpoint

REDO = empty

For each entry in the log, starting at the last checkpoint

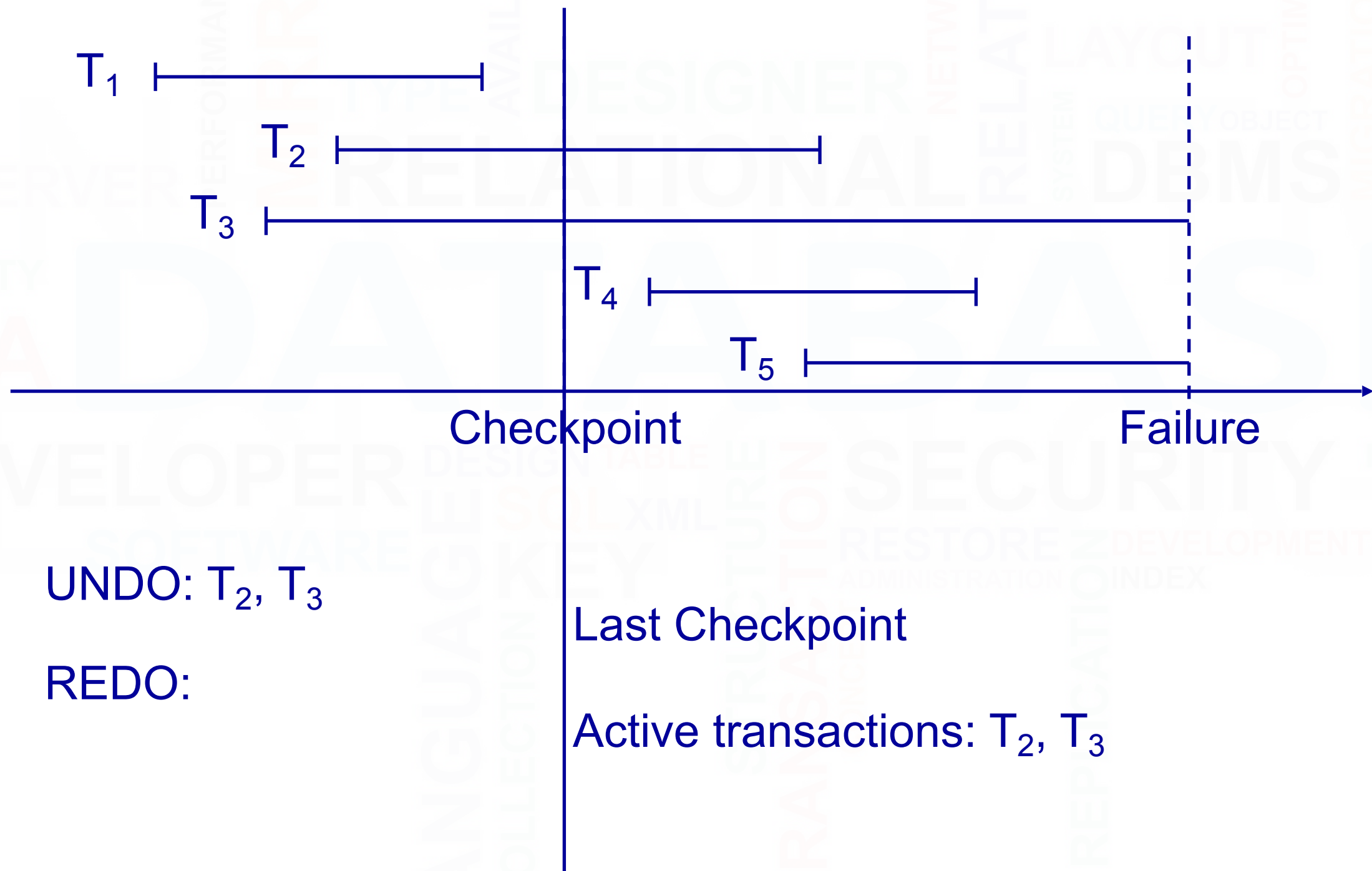
- If a BEGIN TRANSACTION entry is found for T

 - Add T to UNDO

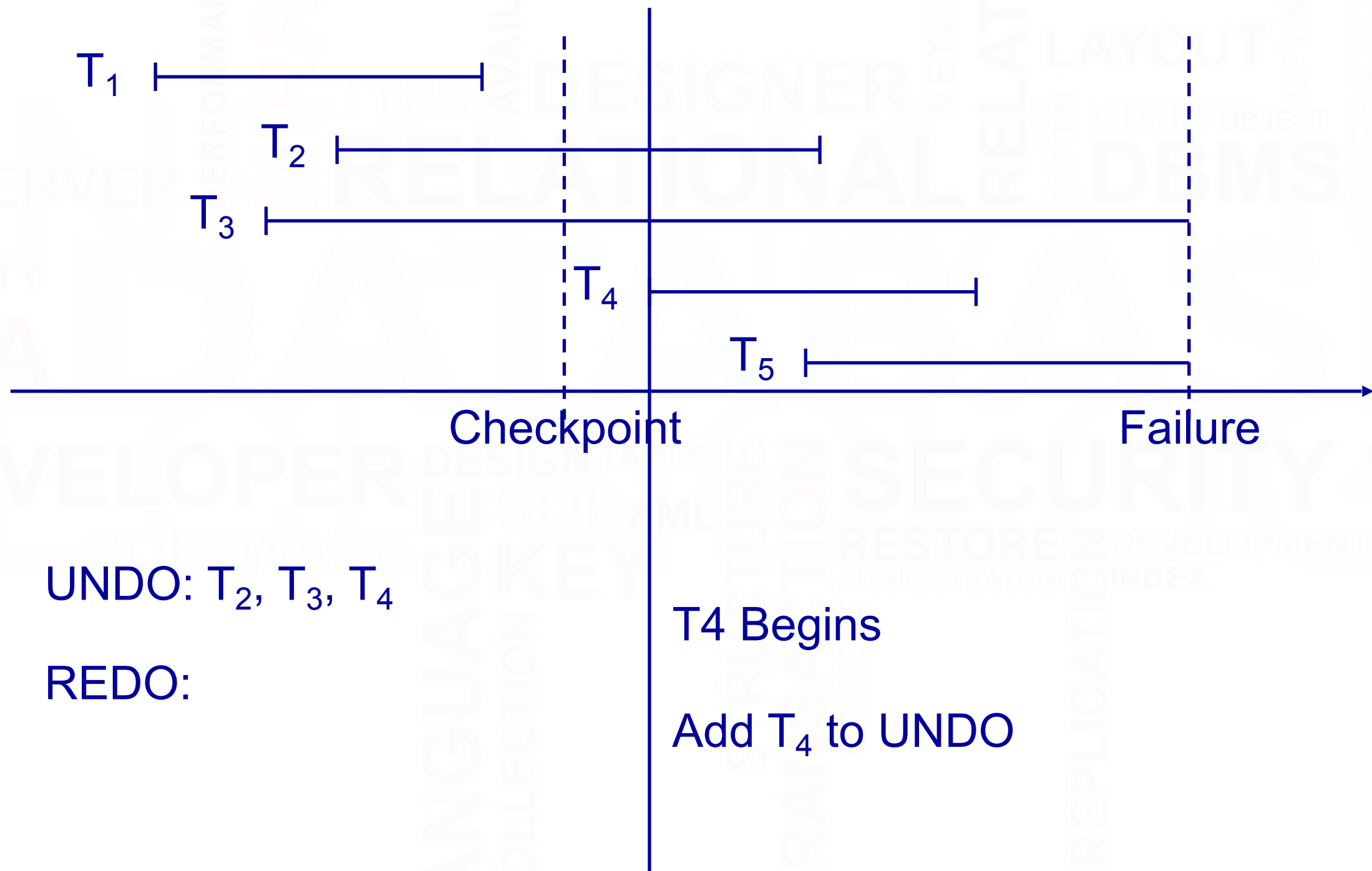
- If a COMMIT entry is found for T

 - Move T from UNDO to REDO

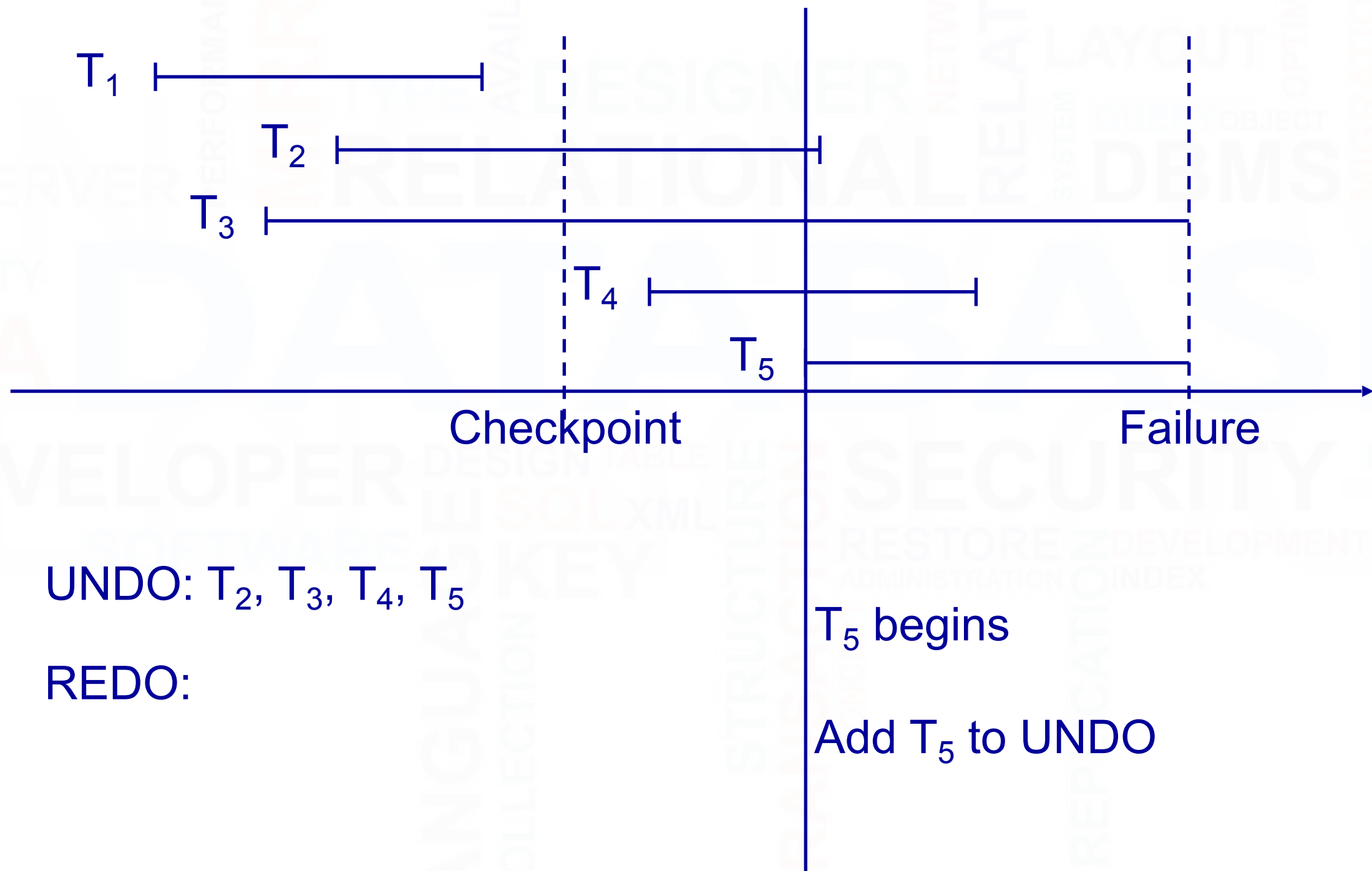
TRANSACTION RECOVERY



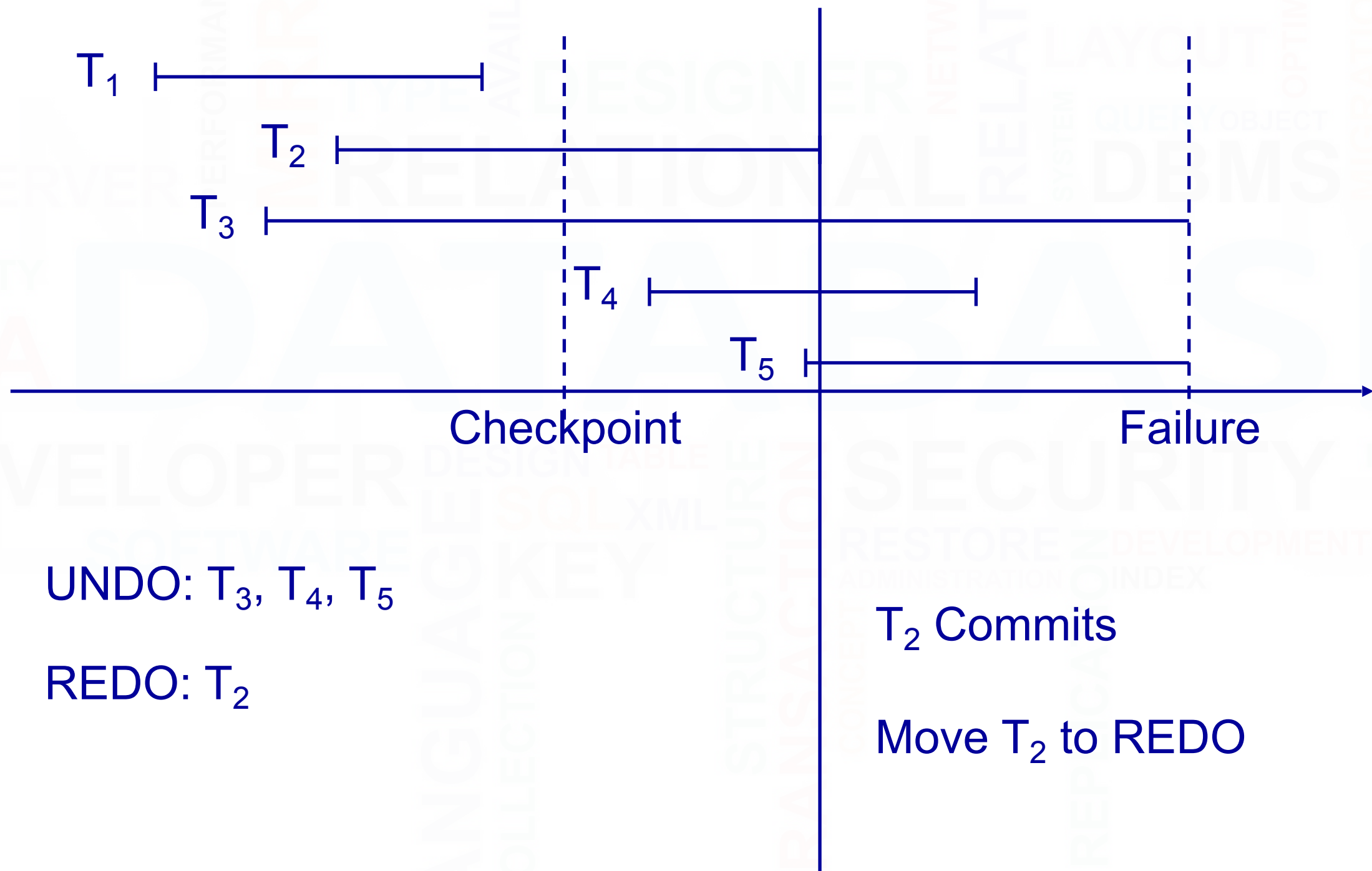
TRANSACTION RECOVERY



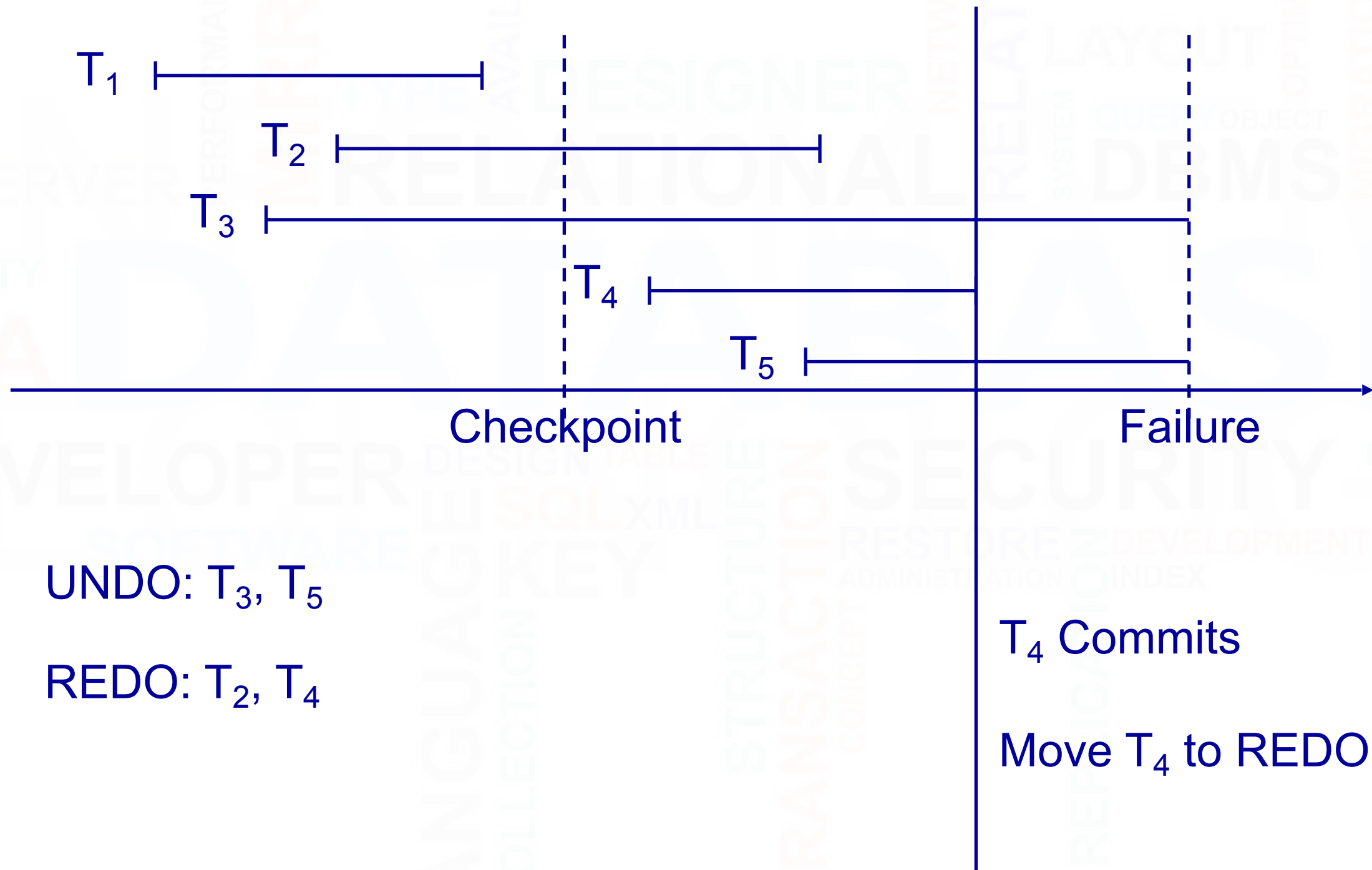
TRANSACTION RECOVERY



TRANSACTION RECOVERY



TRANSACTION RECOVERY



FORWARDS AND BACKWARDS

➤ Backwards recovery

- We need to undo some transactions
- Working backwards through the log we undo any operation by a transaction on the **UNDO** list
- This returns the database to a consistent state

➤ Forwards recovery

- Some transactions need to be redone
- Working forwards through the log we redo any operation by a transaction on the **REDO** list
- This brings the database up to date

MEDIA FAILURES

- System failures are not too severe
 - Only information since the last checkpoint is affected
 - This can be recovered from the transaction log
- Media failures (disk crashes etc) are more serious
 - The data stored to disk is damaged
 - The transaction log itself may be damaged

BACKUPS

- Backups are needed to recover from media failure
 - The transaction log and entire contents of the database is written to secondary storage (often tape)
 - Time consuming, and often requires down time
- Backups frequency
 - Frequent enough that little information is lost
 - Not so frequent as to cause problems
 - Every day (night) is common
- Backup storage



RECOVERY FROM MEDIA FAILURE

- 1. Restore the database from the last backup
- 2. Use the transaction log to redo any changes made since the last backup
- If the transaction log is damaged you can't do step 2
- Store the log on a separate physical device to the database
- The risk of losing both is then reduced

CONCURRENCY

- Large databases are used by many people
 - Many transactions to be run on the database
 - It is desirable to let them run at the same time as each other
 - Need to preserve **isolation**
- If we don't allow for concurrency then transactions are run sequentially
 - Have a queue of transactions
 - Long transactions (eg backups) will make others wait for long periods

CONCURRENCY PROBLEMS

- In order to run transactions concurrently we interleave their operations
- Each transaction gets a share of the computing time
- This leads to several sorts of problems
 - Lost updates
 - Uncommitted updates
 - Incorrect analysis
- All arise because isolation is broken

LOST UPDATE

T1	T2
Read(X) $X = X - 5$	Read(X) $X = X + 5$
Write(X)	Write(X)
COMMIT	COMMIT

- T1 and T2 read X, both modify it, then both write it out
- The net effect of T1 and T2 should be no change on X
- Only T2's change is seen, however, so the final value of X has increased by 5

UNCOMMITTED UPDATE

T1	T2
Read(X) X = X - 5 Write(X)	Read(X) X = X + 5 Write(X) COMMIT
ROLLBACK	

- T2 sees the change to X made by T1, but T1 is rolled back
 - The change made by T1 is undone on rollback
 - It should be as if that change never happened

INCONSISTENT ANALYSIS

T1	T2
Read(X) X = X - 5 Write(X)	
Read(Y) Y = Y + 5 Write(Y) COMMIT	Read(X) Read(Y) Sum = X+Y

- T1 doesn't change the sum of X and Y, but T2 sees a change
- T1 consists of two parts – take 5 from X and then add 5 to Y
- T2 sees the effect of the first, but not the second

END

This is Not! the End of the DB Course

The course will be continues next week!